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 compound data, 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 on its own.

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 on all academic matters and his support in both research and life.

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

Thanks to Tom Van Cutsem, Denis Carnier, Filip Strömbäck, Pontus Haglund, Edwin Florez Gomez, David Bremner, John Clements, Srikumar Subramanian, Kimball Germane, Lukas Ahrenberg, Otto Seppälä, Juha, and Kathi for using or attempting to use my systems in their courses and for sharing datasets and feedback.

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

Thanks to the students and friends who have used my systems. Your feedback has been encouraging and insightful.

Thanks to Lumeng for companionship, happiness, and love.

Thanks to my parents for their unconditional support throughout my life and academic journey.

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

Thanks to NSF for partially funding this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

1.1 What Is 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 values (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 [30]) and deviation (such as the documented semantic oddities of JavaScript and Python [58, 24, 34, 35]). 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 also be non-trivial to understand. CS-majored students and even professional programmers do not understand these behaviors well. As I will discuss in a related work section (Section 9.4), multiple researchers, in different countries and different kinds of post-secondary educational contexts, have studied how students fare with scope and state. They consistently find that even advanced students have difficulty with such brief programs involving SMoL topics.

Not understanding SMoL can lead to costly consequences: Related bugs can easily take hours to fix; Students not understanding SMoL can not 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 My Contributions to Teaching SMoL

SMoL is a cross-language concept. To teach it effectively, we need to present essentially the same program in multiple languages. In Chapter 3, I present artifacts that enable this multilingual presentation: a core language for presenting programs behave according to SMoL, **the SMoL Language**, and the **SMoL Translator**, which translates from the SMoL Language to several commonly used programming languages. To avoid confusing SMoL itself as defined at Section 1.1 with the SMoL Language, I sometimes refer to the former as **SMoL Characteristics**.

I developed **Stacker**, a tool for tracing the execution of SMoL programs (Chapter 7).

I also built **SMoL Tutor**, an interactive, self-paced tutorial designed to correct misunderstandings about SMoL Characteristics. The Tutor draws on concepts from cognitive and educational psychology to explicitly identify and address misconceptions. Chapter 8 presents the tutor and related studies.

Throughout my research, I have identified several common SMoL misconceptions. Chapter 6 presents the process and results. Misconceptions are closely related to a *new* idea, misinterpreters, which is presented in Chapter 5.

Other chapters support these contributions: Chapter 4 describes the study populations, while Chapter 11 provides an overall discussion and conclusion.

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

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 Programming Language Concepts

In the programming languages (PL) community, **semantics** is a formal way of describing meaning. There is a distinction between the meaning of a program and the meaning of a programming language. The semantics of a programming language defines the meaning of all its programs—essentially, a function that maps any program to its meaning.

Some programming languages only allow execution as the primary interaction with a program. Others (e.g., Java) enable static checking, which provides meaning to programs before execution. To distinguish these two forms of meaning, the PL community uses the terms **statics** (for static checking) and **dynamics** (for execution semantics) [26]. This work focuses on dynamics.

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 [26]. Another approach is definitional interpreters, where an interpreter—a program that executes other programs—serves as the semantic definition of the language.

This dissertation is concerned with teaching dynamics.

2.2 Misconceptions and Mistakes

A **misconception** is an incorrect mental model. A **mistake**, on the other hand, is an incorrect action. Mistakes do not necessarily indicate misconceptions because they might happen for various random reasons (e.g., a typo or a misclick). However, if students consistently make the same mistake or provide reasoning that suggests a flawed understanding, then we can infer a misconception.

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

2.3 Tracing

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

Tracing can be done in different ways:

- Automatically, using tools like debuggers or Python Tutor [25].
- Manually, using pen and paper [54].
- Manually within a computer environment, using interactive tools [11].
- A mix of the above, where students manually interact with an automated system [49].

This dissertation presents a tracing tool and associated learning activities.

2.4 Program Trace Visualization Tools

Existing tools that visualize program traces often refer to themselves as **Program Visualization Tools** (e.g., the visualization tool by Cai et al. [6], Python Tutor [25], UUhistle [50], VILLE [41], Jeliot 3 [32]).

Although this term is widely used, I will refer to these tools—along with Stacker, which will be introduced in this dissertation—as **Program Trace Visualization Tools**. As Rajala et al. [41] pointed out, program visualization tools do not necessarily depict program traces:

> 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. (Italic formatting from the original text.)

Comparing the listed tools to static program visualization tools is not particularly meaningful, as they present fundamentally different types of information.

2.5 Notional Machines

The term "Notional Machine" was probably first introduced by Du Boulay, O'Shea, and Monk [16], who defined it as "the idealized model of the computer implied by the constructs of the programming language." Later, Du Boulay [15] 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 [15] 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 [42, 48]) typically restrict the term to semantics. However, as Duran, Sorva, and Seppälä [17] noted, the term is often used inconsistently. Many papers labeled as "notional machine" research actually focus on tracing, RPBs, or misconceptions.

To avoid confusion, this dissertation minimizes the use of the term "notional machine."

2.6 Rules of Program Behavior

Traditional semantics is often presented in highly formal mathematical notation or lengthy language specifications. While suitable for semanticists, these representations are rarely accessible to students or even instructors without a formal PL background.

Duran, Sorva, and Seppälä [17] examine the role of semantics-related knowledge in CS education. They emphasize that teachers must convey learning objectives effectively and translate them into instructional materials. According to them, these objectives should explicitly describe "how computer programs of a particular kind behave when executed." They refer to such objectives as **rules of program behavior (RPBs)**. Following their approach, I define my RPBs in Chapter 8.

CHAPTER 3

REPRESENTING SMOL

Teaching SMoL requires students to read and understand computer programs, which in turn requires a programming language. Ideally, we need 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**, along with the **SMoL Translator**, a tool that translates SMoL programs into several commonly used programming languages.

Rather than reusing an *existing* language, I designed a *new* one because while SMoL is a representative model, no single existing language fully aligns with SMoL. For example, Java restricts closure-referenced variables [30], and Python employs an unusual scoping rule [35].

- Section 3.1 details the SMoL Language.
- Section 3.2 describes the SMoL Translator and its key design choices.
- Section 3.3 provide a broader discussion.

3.1 The SMoL Language

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 condition (e) and a body. For convenience, I collectively refer to `program`s and `body`s 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 in which I conducted my earlier study. However, 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 3.2) for processing SMoL programs.

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 typed language (e.g., Java's `Object` type).
- "Vectors", also known as *arrays*, are fixed-length mutable data structures.
- "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.

```
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 ...
```

Figure 3.1: The syntax of 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

Table 3.1: Primitive operators in the SMoL Language

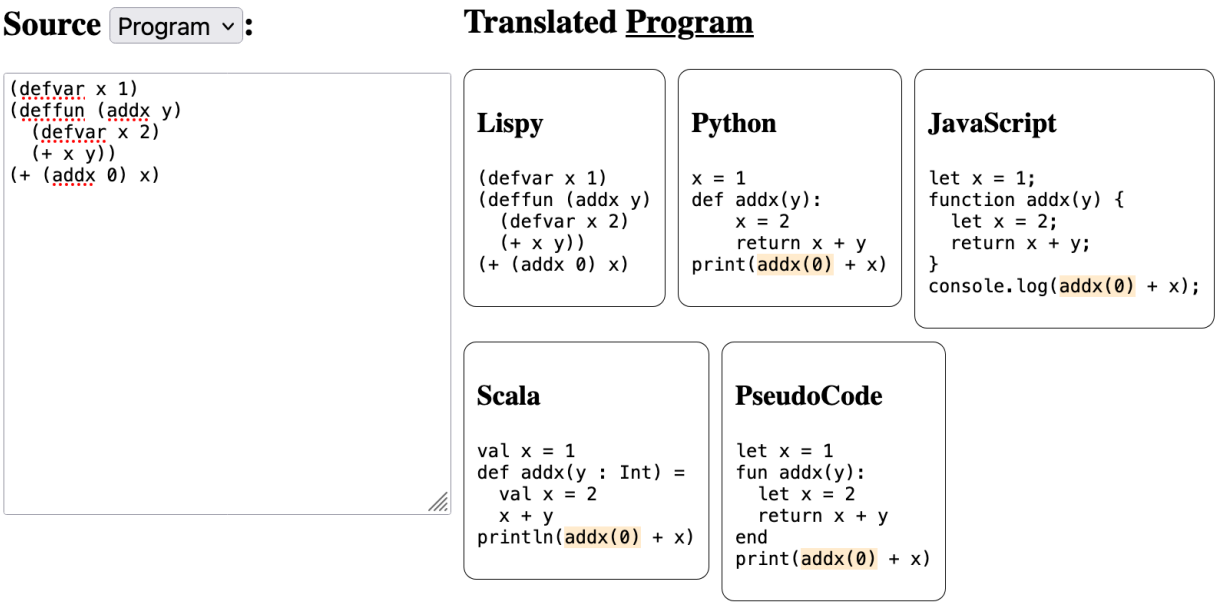


Figure 3.2: SMoL Translator Web App. Showing a translated program with the cursor hovering on one of the `addx(0)`.

3.2 The SMoL Translator

Figure 3.2 illustrates the UI of the **SMoL Translator**, which translates SMoL programs to JavaScript, Python, Scala 3 [WIP], and a pseudocode syntax. The translator is available as a webpage at

smol-tutor.xyz/smol-translator

and as an `npm` package at

github.com/brownplt/smol-translator

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 3.2.1 discusses design decisions that are broadly applicable, affecting many supported target languages or commonly used programming languages.
- Section 3.2.2 focuses on design decisions likely specific to JavaScript.
- Section 3.2.3 focuses on design decisions likely specific to Python.
- Section 3.2.4 focuses on design decisions likely specific to Scala.

3.2.1 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 every infix operation that appears immediately inside another infix operation.
3. Parenthesize an infix operation only when ambiguity exists.

The SMoL Translator implements solution 2. The issue with solution 1 is that it outputs programs like `f((a + b))`, which appear non-idiomatic. Solution 2 works well in my experience and requires considerably less engineering effort than solution 3.

Solution 3 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.

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.

Insert "return"s

The SMoL Language dictates that functions return the last expressions in their body. However, many languages uses explicit "return" statements. When translating to those languages, the translator must apply "return" statements appropriately.

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.

Insert "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 (STLC). There is no strong rationale behind this design. An alternative design could involve making the SMoL Language typed, which is discussed in Section 3.3.

Mutability of Variables

In the SMoL Language, all variables are defined in the same way, regardless of whether the programmer intends to mutate them. However, many languages distinguish between mutable and immutable variables, and it is more idiomatic in those languages to specify whether a variable is 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.

3.2.2 Design Decisions Specific to JavaScript

JavaScript is known for its reluctance to raise errors. Consider the following SMoL program, which results in an error:

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

Its 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 does it raise an error when dividing by zero.

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

3.2.3 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`" and inserts `nonlocal` or `global` declarations to resolve scope correctly. Note that this translation does *not* always preserve the semantics. For instance, the following SMoL program results in an error because it assigns to an undefined variable, but its Python version does not:

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

3.2.4 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.

3.3 Discussion

The SMoL Language 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.

3.3.1 Desired Properties for Multilingual Presentation

There are several desired properties for multilingual presentation:

Straightforwardness Easy connections among programs.

Totality A program exists in every language of interest.

Semantics preservation The programs should produce essentially the same output.

Idiomaticity Every program should look idiomatic in its target language.

The system works well 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 3.2.1 and Python `lambda` in Section 3.2.3).
- **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", data types in Section 3.2.1, JavaScript's error problems in Section 3.2.2, and Python scope in Section 3.2.3)
- **Idiomaticity** is compromised when the target language uses infix syntax, declares mutable and immutable variables differently, has nonstandard scoping rules, or has nonstandard conventions for writing argument lists (see infix operation and mutability of variables in Section 3.2.1, Python scope in Section 3.2.3, and Scala's argument list problem in Section 3.2.4)

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 this property critical for a multilingual learning experience.

3.3.2 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).

3.3.3 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.

3.3.4 Miscellaneous

If the SMoL Language does not automatically print top-level expressions, we can preserve semantics for more programs without any significant downsides. This is a potential area for future work.

I do not plan to change how the SMoL Translator handles infix operations or its lack of precision regarding mutability of variables, as explained in Section 3.2.1.

I do not intend to modify the system to accommodate the individual behaviors of languages discussed in Sections 3.2.2 to 3.2.4, as these changes would likely degrade the system's suitability for other languages.

CHAPTER 4

STUDIED POPULATIONS

The vast majority of this work is done with students in a "Principles of Programming Languages" course at a selective, private US university (hereafter **the PL course** or **University 1**). The course has about 70–75 students per year. It is not required, so students take it by choice. Virtually all are computer science majors. Most are in their third or fourth year of tertiary education; about 10% are graduate students. All have had at least one semester of imperative programming, and most have significantly more experience with it. Most have had close to a semester of functional programming. The student work described here was required, but students were graded on effort, not correctness.

One study was conducted in an accelerated introductory course at University 1 (hereafter, the **Accelerated Intro course**). Some of the students had prior experience with computer programming. In addition, to enroll in the course, students were required to read *How to Design Programs* [19] (HtDP) and successfully complete a number of Racket programming exercises.

Some studies are repeated at other populations:

University 2 is from a primarily public university in the US. The university is one of the

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

Course at a Belgium University The university is a major research university in Belgium. [FILL]

Project-based PL Course at a Sweden University The university is a prominent research institution in Sweden. The students have taken a Python course previously. They then take this course. It is a project-based course where students mostly create simple programming languages in Ruby. There is a course that covers some theory beforehand (e.g. lexing, parsing, etc.). The project course then covers some topics on how to structure compilers: 4 lectures on grammars, lexical analysis, variable scoping, etc. The study was mandatory in the course. It was published at the beginning of the course, and students were recommended to finish it by the end of the first half of the semester, before they were expected to start implementing their languages.

Online Population A separate instance of the Tutor was published on the website of a programming languages textbook [46]. Over the course of 8 months, 597 people started with the first module and 103 users made it to the last one. To protect privacy, I intentionally do not record demographic information, but I conjecture that the population is largely self-learners (who are known to use the accompanying book), including some professional programmers. It is extremely unlikely to be the students from either university, because they would not get credit for their work on the public instance; they needed to use the university-specific instance. Furthermore, since they were not penalized for wrong answers, it would make little sense to do a "test run" on the public instance. Finally, I note that there is no overlap between the dates of submission on

the public instance and the semester at the university of **P1**.

These other populations are at least somewhat different from the original population, and help me assess whether the problems I identify are merely an artifact of the first population.

CHAPTER 5

MISINTERPRETERS

5.1 Misinterpreters Model Misconceptions

A **misinterpreter** is an interpreter that executes programs incorrectly. Misinterpreters are closely related to misconceptions, which are patterns of mistakes (Section 2.2). In the context of understanding program outputs, a mistake is an incorrect *pair* from a program to an output. A misconception, then, can be understood as a systematic pattern of such incorrect pairs, i.e., 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. For readers with a Programming Languages background, a misinterpreter is essentially a definitional interpreter for a misconception.

5.2 Misinterpreters Help Identify and Correct Misconceptions

Beyond offering precise definitions, misinterpreters also help identify misconceptions. Consider the following SMoL program:

```
(defvar x 0)
(defvar x 1)
(print x)
```

This program results in an error because the SMoL Language, like many languages, does not allow a variable to be defined twice in the same block. However, some people might believe this program outputs `1`, reasoning that "the second `x` redefines the first `x`." If we use this program to identify misconceptions, we might conclude that those who answer `1` hold a redefinition misconception.

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

- Shadowing misconception: The second `x` *shadows* the first `x`.
- Mutating misconception: The second `x` *mutates* the first `x`.

To distinguish these misconceptions, we can ask people to interpret a slightly different program:

```
(defvar x 0)
(deffun (f)
  x)
(defvar x 1)
(print (f))
```

Under the shadowing misconception, this program would output `0`; under the mutation misconception, it would output `1`.

This example highlights several ways in which misinterpreters can help address misconceptions:

Clarifying Ambiguity in Misconception Descriptions Given a wrong answer and a natural language description of a related misconception, misinterpreters help uncover ambiguities in the description. In the example above, we might initially describe the misconception as involving "redefinition." However, when implementing the misinterpreter, we must decide whether earlier definitions are shadowed or mutated, revealing the underlying ambiguity in the term "redefine".

Validating Mistake-Misconception Mappings Given a wrong answer and a set of known misconceptions, misinterpreters help determine whether the wrong answer uniquely maps to a specific misconception by running the program through different misinterpreters. In the example above, if we already know about both the shadowing and mutating misconceptions, we could run the first program through both misinterpreters and discover that the wrong answer could result from multiple misconceptions. Conversely, if a wrong answer maps to no existing misinterpreters, this suggests the presence of a new misconception.

Designing Diagnostic Programs Given known misconceptions, misinterpreters assist in designing programs that can effectively detect them. The example above illustrates an iterative refinement process: we propose a program, observe that some wrong answers correspond to multiple misconceptions, refine the program, and validate it until we achieve a program that differentiates misconceptions more clearly. This process can, in principle, be automated using *mutation testing* or even extended through *program synthesis*, generating diagnostic programs without requiring an initial design.

Misinterpreters are also useful for **generating targeted feedback**. Given a wrong answer and a known misconception, misinterpreters enable misconception-aware feedback. Chapter 8 discusses a case where wrong answers are drawn from a predefined set, allowing feedback

to be prepared in advance. Chandra et al. [8] present a more technically complex scenario where wrong answers can be arbitrary, requiring feedback to be computed dynamically rather than pre-scripted.

CHAPTER 6

SMOL MISCONCEPTIONS

Section 6.2 lists the misconceptions I identified.

Section 6.3 discusses related work on identifying misconceptions, and summarize approaches that are known to be good or bad.

Section 6.4 describes how I identified my collections of misconceptions.

6.1 Misconceptions Identified by Prior Work

[9]

[40]

[49]

[28]

[15]

What Exactly are Program Language Behavior Misconceptions?

They are Program Language Behavior Conceptions that are different from the correct program language behavior conception.

The correct conception is the prefect understanding of the *relation* Program $\times$ Input $\times$ Output, subject to limitation of human cognition. For example, if a person have the correct conception, they should be able to, given a program and an input, predict the output, but they might or might not be able to do so when the program's source code is very long. The above example requires only an understanding of the *function* Program $\times$ Input $\rightarrow$ Output. The word *relation*, which includes more than the function, was chosen intentionally. A person with the correct understanding should also be able to, subject to the constraint of human cognition, do many other kinds of tasks, for example:

Test-Driven Development Given input-output pairs, construct a program that map those inputs to the corresponding outputs.

Proof Equivalence of Program Fragments Given two program fragments, judge if they don't change the input-output mapping for all program context.

Debug Given a program and some input-output pairs, modify the program so that it matches the input-output pairs.

Construct Classifiers Given two programs, construct an input that the programs map to different outputs.

Conversely, if a person has a misconception, they will still be able to do the tasks but can not do them perfectly even within the range of that person's cognition capacity.

Notably, if a person can't do the tasks at all (e.g., getting completely lost), that person neither has the correct conception nor has a misconception. Rather, that person is lacking

a conception.

6.2 Misconceptions That I Identified

Tables 6.1 to 6.3 presents the misconceptions that my data suggest. All the tables follow the same format: Each table row defines a misconception and provides an example program; Each example program comes with the correct output and the incorrect output that corresponds to the misconception.

The **misinterpreters**, i.e., the definitional interpreters for the misconceptions, are provided in Appendix A.

6.3 How (Not) to Identify Misconceptions?

6.3.1 We should overcome expert's blind spot

In Section 9.4, I discuss several papers that have provided reports of student misconceptions with different fragments of SMoL. However, it is difficult to know how comprehensive these are. While some are unclear on the origin of their programs, they generally seem to be expert-generated.

The problem with expert-generated lists is that they can be quite incomplete. Education researchers have documented the phenomenon of the *expert blind spot* [33]: experts simply do not conceive of many learner difficulties. Thus, we need methods to identify problems beyond what experts conceive.

Finally, I am inspired by the significant body of education research on *concept inventories* [27] (with a growing number for computer science, as a survey lists [52]). In terms of mechanics, a concept inventory is just an instrument consisting of multiple-choice questions (MCQs), where each question has one correct answer and several wrong ones. However, the wrong ones are chosen with great care. Each one has been validated so that if a student

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 6.1: Aliasing-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 6.2: Scope-related 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.	`(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`

Table 6.3: Miscellaneous misconceptions

picks it, we can quite unambiguously determine *what misconception the student has*. For instance, if the question is "What is `sqrt(4)`?", then `37` is probably an uninteresting wrong answer, but if people appear to confuse square-roots with squares, then `16` would be present as an answer.[1]

6.3.2 We should define misconceptions precisely

We'd better give identified misconceptions accurate descriptions using misinterpreters Chapter 5. As discuss in Section 9.4, although misconceptions are commonly documented as brief statements (and occasionally with a couple of programs for illustration), this practice makes it rather difficult to tell if two misconceptions are the same misconceptions.

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

6.4 How I Identified the Misconceptions

The considerations discussed in Section 6.3 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 have a sense of what misconceptions students have, and
3. We can generally associate wrong answers with specific misconceptions, approaching a concept inventory.

6.4.1 Generating Problems Using Quizius

My main solution to the expert blind spot is to use the Quizius system [44]. 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 [29] and uses that to choose a program.

[2]In the course (Chapter 4), 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.

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.

After running Quizius in the course (Chapter 4), I took over as an expert. Determining which is the right answer is easy. Where expert knowledge is useful is in *clustering the rationales*. If all the rationales for a wrong answer are fairly similar, this is strong evidence that there is a common misconception that generates it. If, however, there are multiple rationale clusters, that means the program is not discriminative enough to distinguish the misconceptions, and it needs to be further refined to tell them apart. Interestingly, even the correct answer needs to be analyzed, because sometimes correct answers do have incorrect rationales (again, suggesting the program needs refinement to discriminate correct conceptions from misconceptions).

Prior work using Quizius [44] finds that students do author programs that the experts did not imagine. In my case, I seeded Quizius with programs from prior papers (Chapter 9), 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.

6.4.2 Distill Programs and Misconceptions From Quizius Data

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

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

```
(defvar x 1)
```

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

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

- Some programs relied on (or stumbled upon) intentionally underspecified aspects of SMoL Language such as floating-point versus rational arithmetic. While these are important to programming in general, I considered them outside the scope of SMoL Characteristics (due to their lack of standardization). Mystery languages (Chapter 9) are a good way to explore these features.

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

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

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

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

- The existing programs did not cover all ideas I wanted students to work through. For instance, no Quizius programs alias vectors using function calls. In this case, new programs need to be added to fill in the gap.

- To reduce the number of concepts, I removed programs that relied upon *im*mutable vectors and lists, because they did not seem to create problems. (For brevity, I leave these out of the presentation of SMoL Language in Chapter 3, though they were in the language when I collected the Quizius data.)

- I removed programs of a "Lispy" nature. For example, the following program depended on whether the reader correctly understood this inequality. This checks whether `n` $<$ `3`, but some presumably vocalized it as "greater than `3`, `n`?".

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

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

6.4.3 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 8. 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 6.3. 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.

6.5 p-prim

P-prims are intuition that generally works but might fail on some cases. P-prims themselves are hard to be rejected because they are pre-conceptions. Even if they can be rejected, they probably should not be because they are generally correct. Misconceptions can be a wrong application of some p-prims on certain cases.

CHAPTER 7

STACKER

Section 7.1 provides a guided tour of the Stacker, illustrating its typical usage and introducing its major components.

Section 7.2 delves into the system's details.

Section 7.3 explores the teaching instruments enabled by the Stacker.

Section 7.4 compares the system with similar tools.

Sections 7.5 and 7.6 present two user studies about the Stacker.

Section 7.7 discusses design the design space, summarizing lessons learned from the tool comparison (Section 7.4) and the user studies (Sections 7.5 and 7.6).

7.1 A Guided Tour of Stacker

The Stacker is a GUI application for tracing SMoL programs. Figure 7.1 shows the interface 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

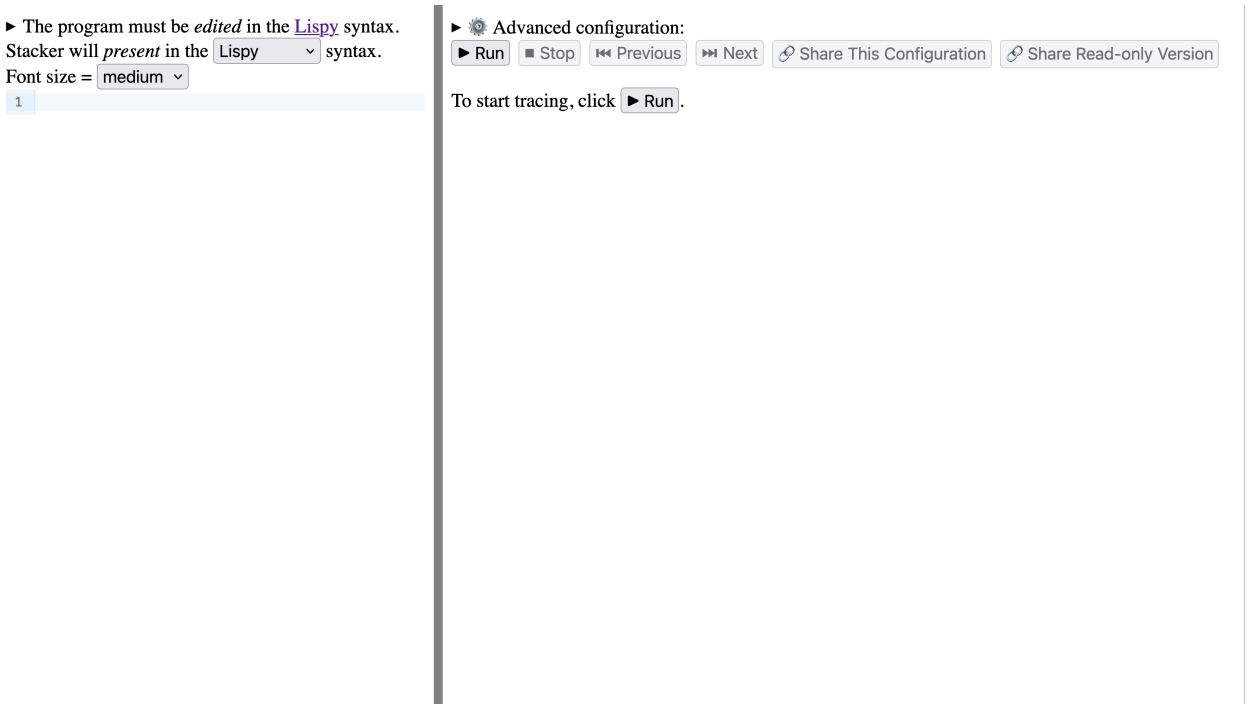


Figure 7.1: The Stacker user interface

top and a program editor, which supports common text-editing features like syntax highlighting. Section 7.2 provides more details.

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 7.2.

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.

7.1.1 Example Program

Figure 7.2 shows the Stacker loaded with the following SMoL program:

```
(defvar x 1)
```

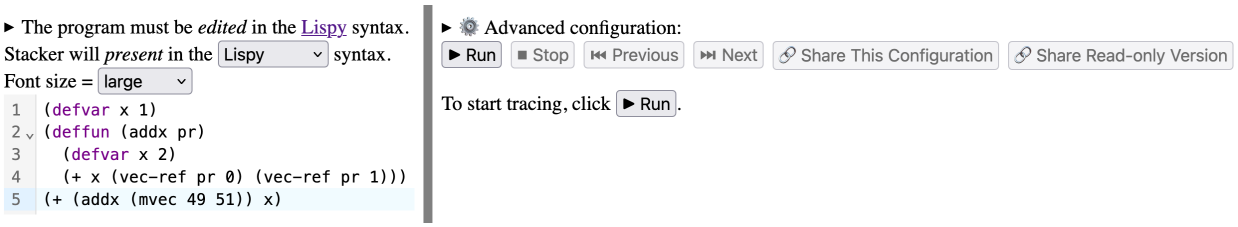


Figure 7.2: The Stacker user interface filled with an example program

```
(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.

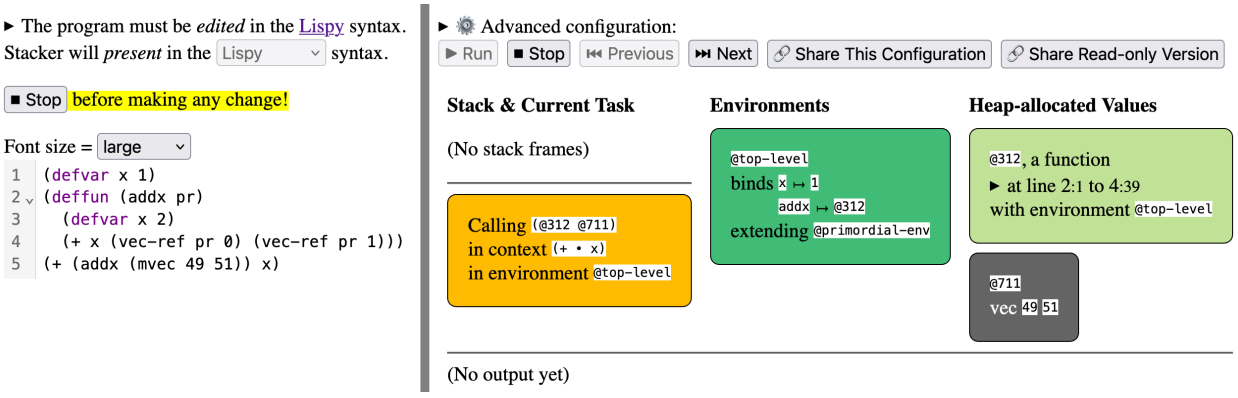


Figure 7.3: An example Stacker trace (showing state 1 out of 5)

7.1.2 Running the Program

Clicking one of the "run" buttons in Figure 7.2 transitions the Stacker to tracing mode Figure 7.3, 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 7.3), 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

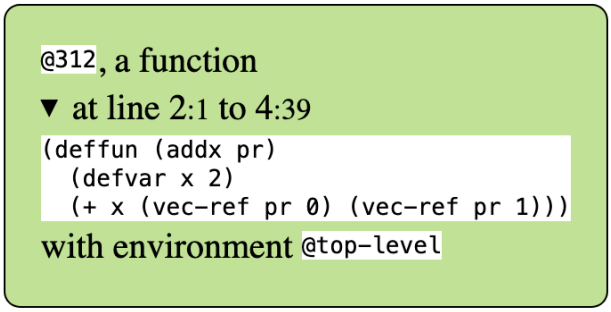
environment of the current task. The box color depends on the kind of tasks. In the current state (Figure 7.3):

- 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.

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 7.3), 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.

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 source code location of a function value reveals its body.

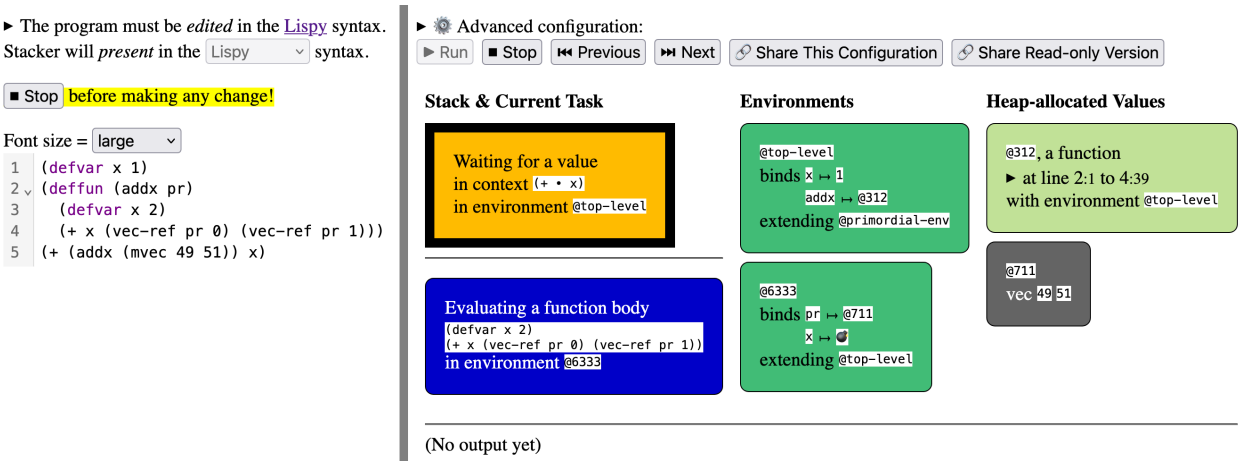


Figure 7.4: An example Stacker trace (showing state 2 out of 5)

Clicking the "Next" button in Figure 7.3 updates the Stacker to Figure 7.4. In Figure 7.3, the Stacker was about to compute a function call, so Figure 7.4 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.

Stack frames are presented as yellow boxes, the same color as function call tasks (Compare Figure 7.3 and Figure 7.4). 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".

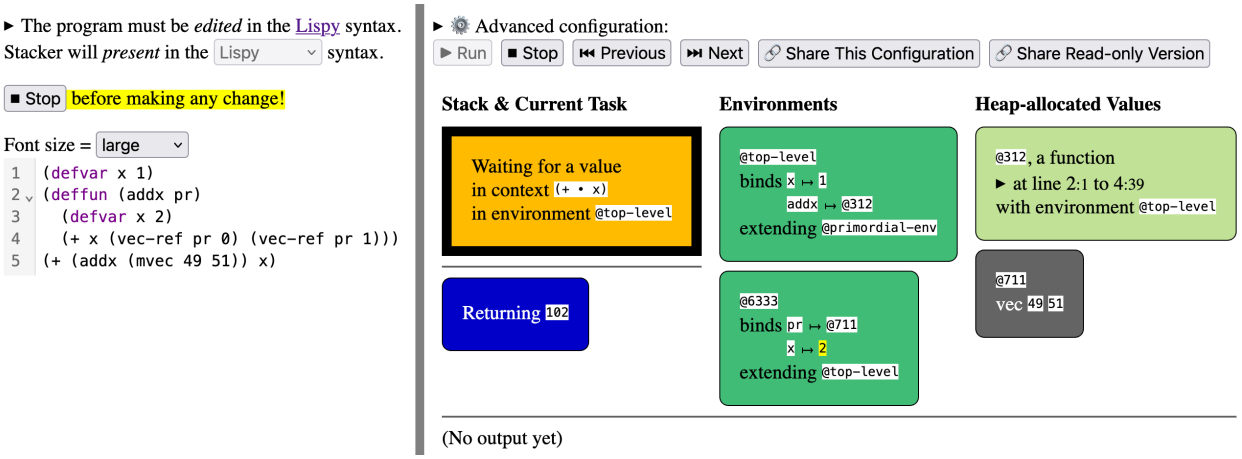


Figure 7.5: An example Stacker trace (showing state 3 out of 5)

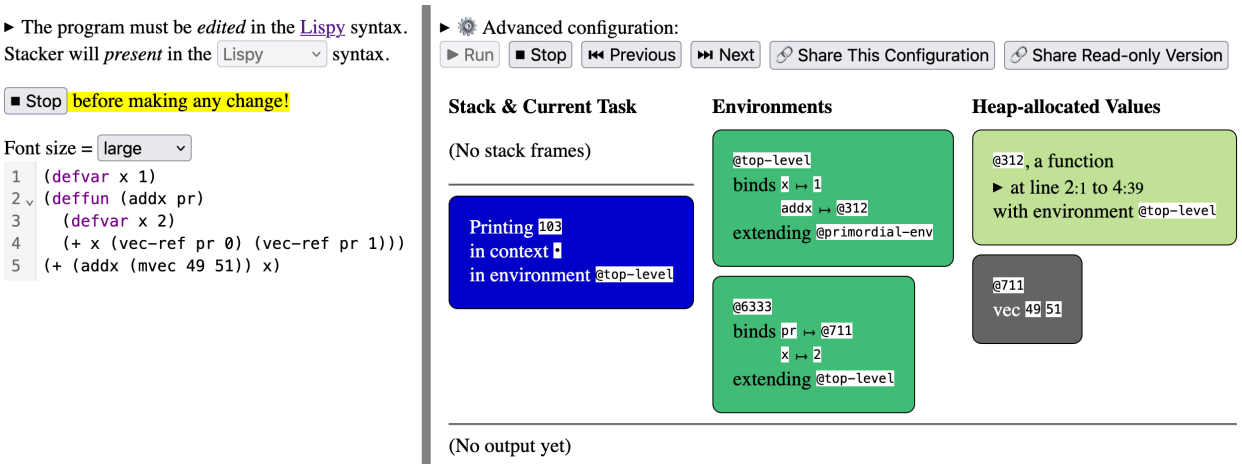


Figure 7.6: An example Stacker trace (showing state 4 out of 5)

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.

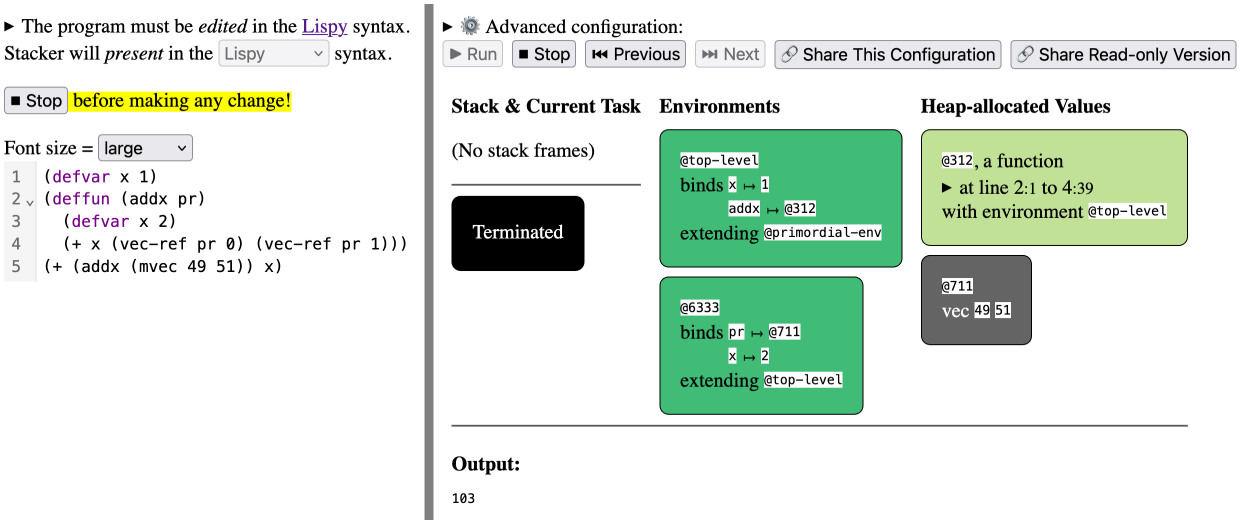


Figure 7.7: An example Stacker trace (showing state 5 out of 5)

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 **??**.

The previous state (Figure 7.3) illustrates the first kind of current task, the current state (Figure 7.4) illustrates the second. The remaining trace states each demonstrate a different task kind:

- Figure 7.5 The stacker is returning from the function call.

- Figure 7.6 The stacker is printing the returned value.

- Figure 7.7 The stacker has terminated.

7.1.3 Key Features of Stacker

I conclude this section with a few highlights on the Stacker:

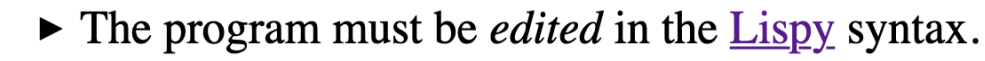
- The Stacker can present the program and the trace in any language supported by the SMoL Translator (Chapter 3), but programs must be edited in the SMoL Language.
- Every (non-final) state presents enough information to predict the next state.
- Users can create a sharable URL for the current state, allowing others to view the exact same trace.
- The Stacker traces are generated on the fly, so traced programs need not be terminating.
- Memory addresses of environments and heap-allocated values are randomly generated, with a controllable random seed.

7.2 More Details on Using Stacker

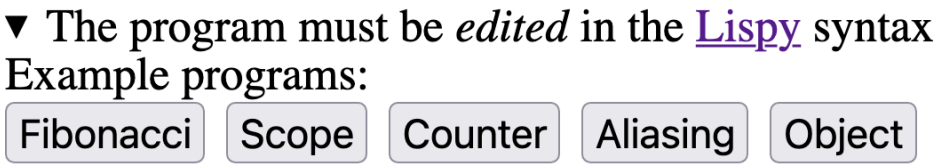
This section presents more details on using the Stacker, complementing the guided tour (Section 7.1).

7.2.1 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 Section 3.1). 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.

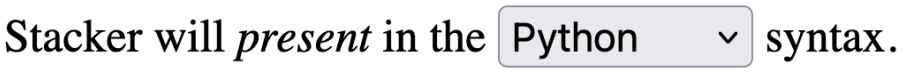
Count defines and tests a "counter" function that increases with each call.

Aliasing aliases data structures in a perhaps confusing way.

Object simulates an object using the SMoL Language.

7.2.2 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 3.2 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 7.8).



7.2.3 Change the Relative Font Size of the Editor

The following dropdown menu controls the font size of the editor, rather than the entire UI. This choice is intentional: since 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:

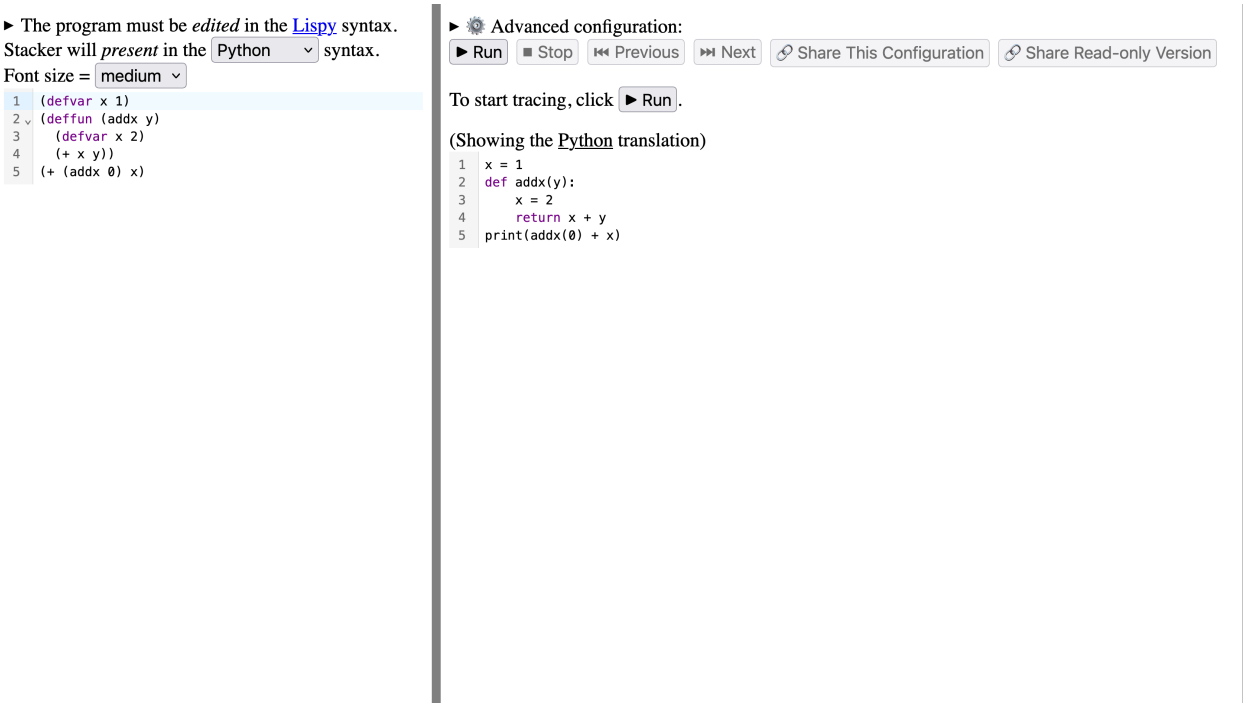
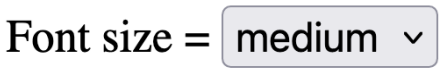


Figure 7.8: The live translation feature of Stacker. When users are editing their programs and the presentation syntax is set to non-Lispy, a live translation of the program is shown on the trace panel.

- 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.



7.2.4 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 [12], which makes the feature straightforward to implement.

7.2.5 Share Buttons

The following buttons generate permanent URLs 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 7.9).

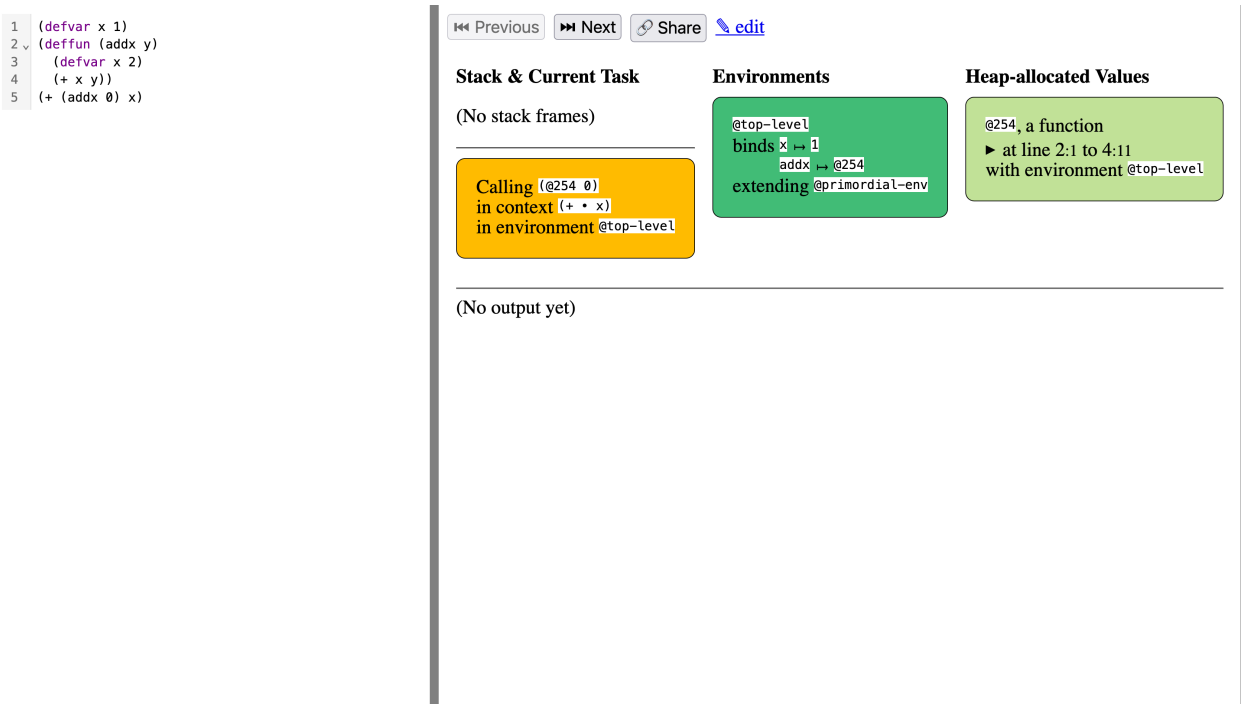


Figure 7.9: A read-only view of a Stacker trace. This kind of view is creatd by "Share Read-only Version" or the "Share" button in a read-only view.

7.2.6 Advanced Configuration

The following UI element provides additional tracing configurations.

▸ ⚙ Advanced configuration:

After expansion, it becomes

▾ ⚙ Advanced configuration:
Random seed = lambda
Hole = •
Print the values of top-level expressions ☑

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.

7.2.7 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.

7.2.8 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.

7.2.9 The Trace Display Area Color-Codes Boxes

The Stacker color-codes boxes according to Table 7.1

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

Table 7.1: All combinations of foreground and background colors in the Stacker

7.3 Educational Use Cases

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.

7.3.1 Predict 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 [38] for a review).

7.3.2 Contrast Two Traces

Many SMoL misconceptions can be view 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.

7.3.3 From State to Value

In the PL course (Chapter 4), students were asked to determine a program's output based on a given trace state (illustrated in Figure 7.10).

This kind of exercise is essentially asking students to predict the final state given an arbitrary state.

7.3.4 From State to Program

In the PL course (Chapter 4), students were also instructed to construct programs that would produce a trace state. Figure 7.11 illusion one of those states. Students were instructed that the solution program always include a `pause` function that returns `0`:

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

In this example, 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)
```

This kind of exercise is essentially asking students to predict the initial state given an arbitrary state.

7.3.5 Using Stacker to Teach Generators

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

Figure 7.10: A state-to-output question

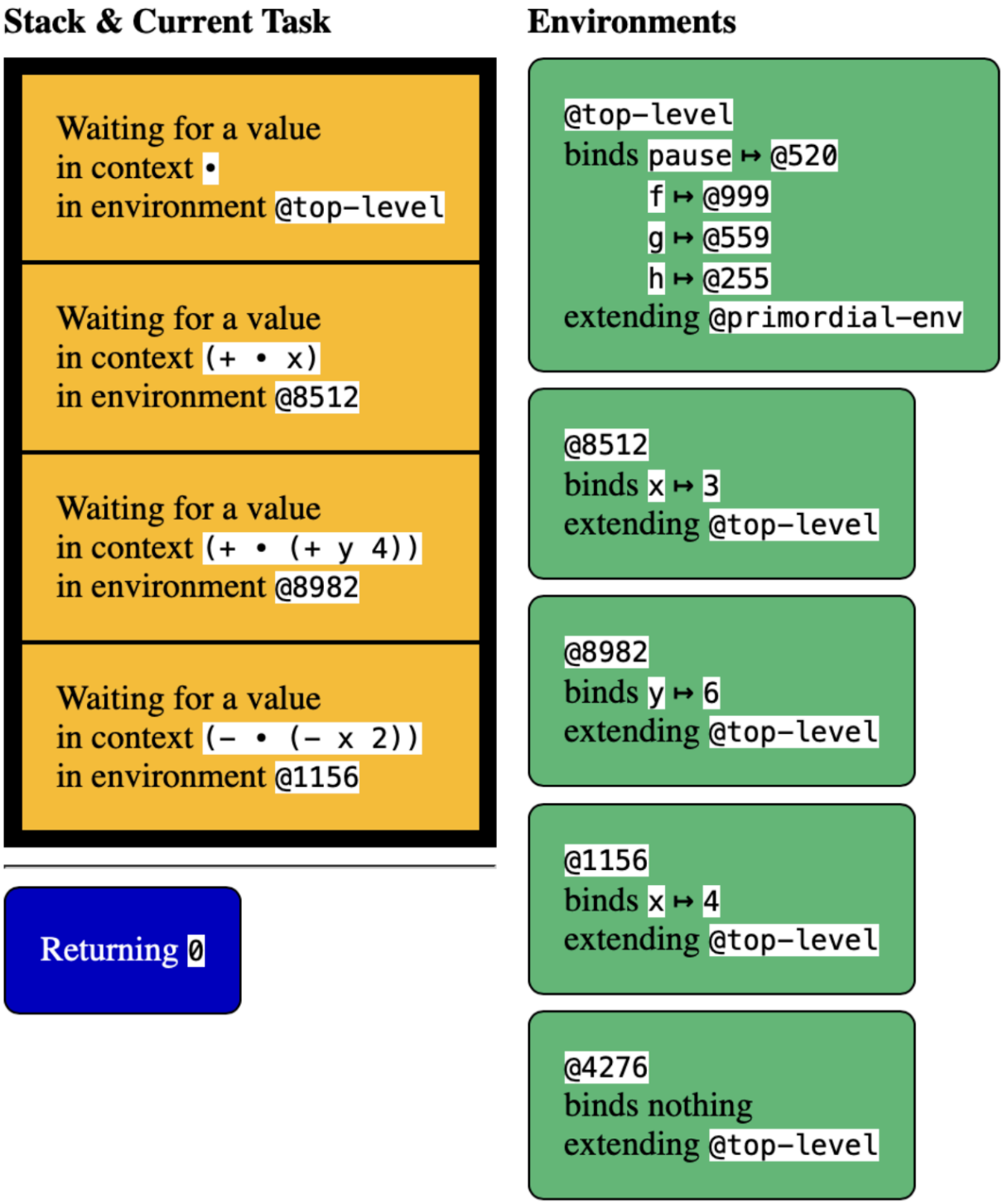


Figure 7.11: A state-to-program question

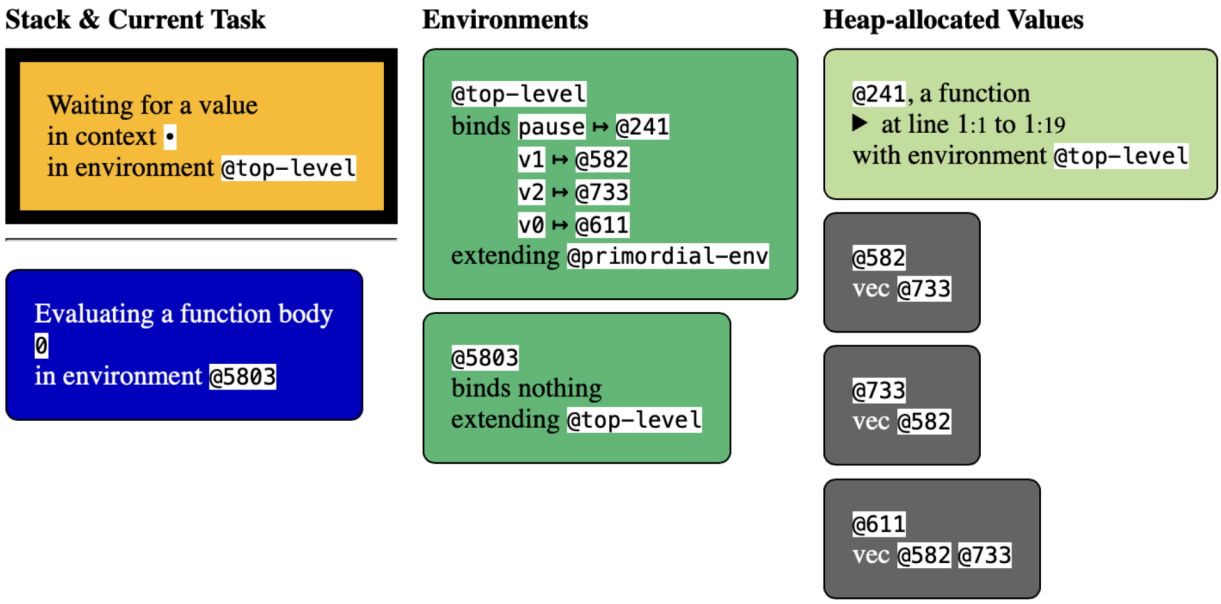


Figure 7.12: A Python program for the Stacker-generator instrument

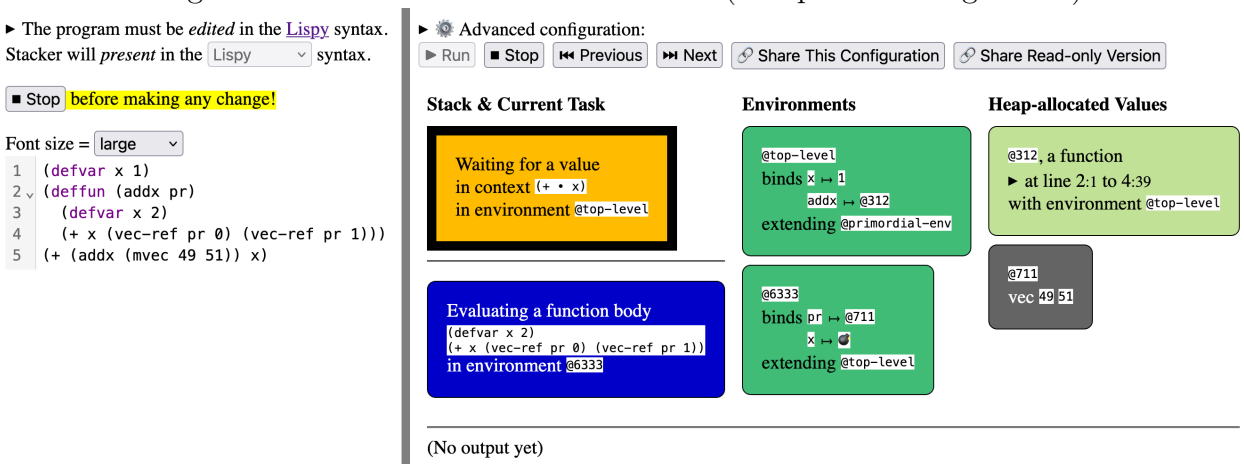
```
def pause():
    return 0

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()
```

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 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 4), students drew Stacker-like diagrams to visualize Python generator behavior. One such program is shown in Figure 7.12.

7.4 Compare Stacker with Other Tools

Figure 7.13: A Screenshot of the Stacker (a duplicate of Figure 7.4)



This section compares the Stacker with several program trace visualization tools by running essentially the same program (Table 7.2) 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 7.13.

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, such as first-class functions and mutable state;

Table 7.2: 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)`

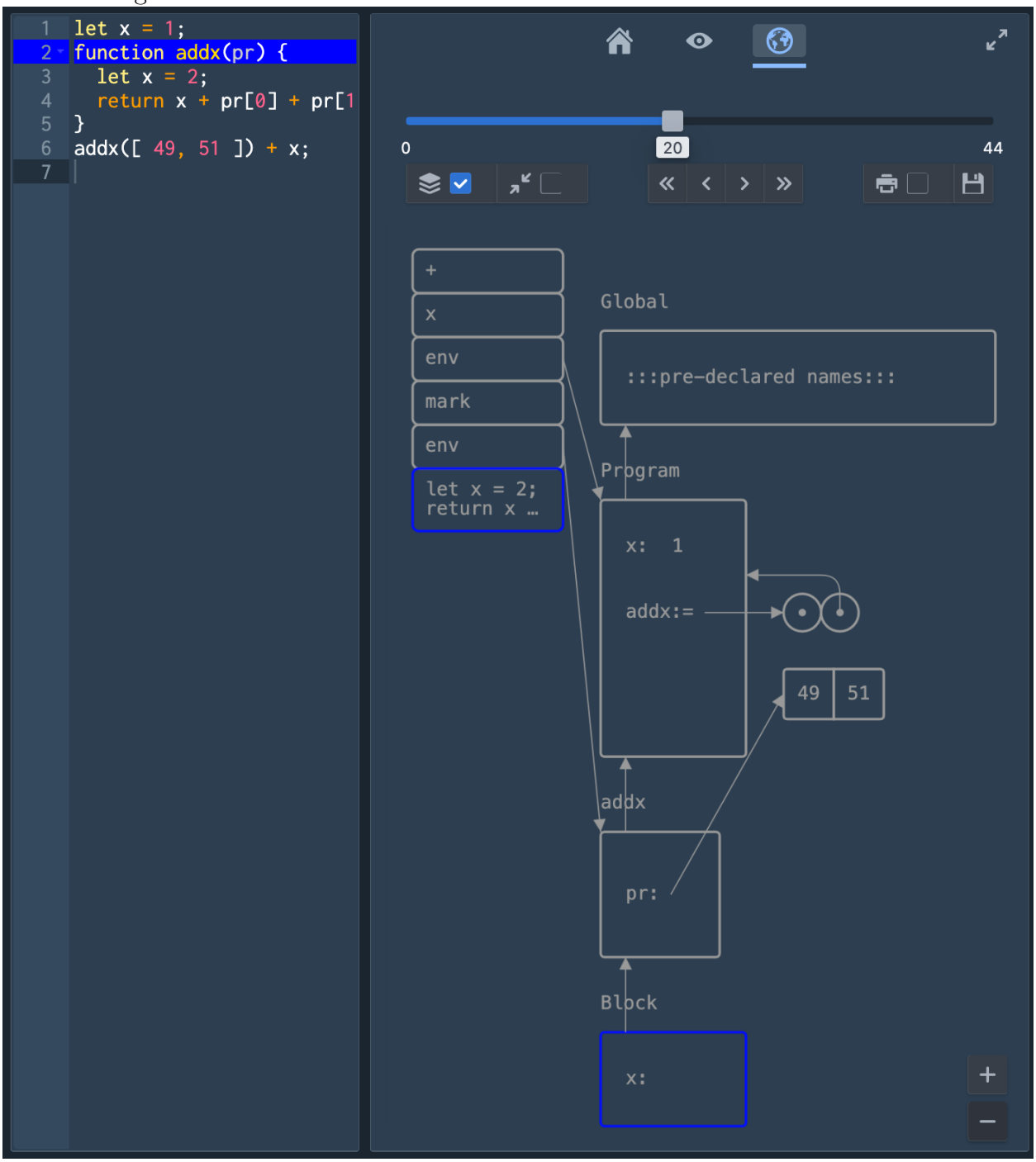
- Cover as many SMoL language features as possible;
- Ensure that each kind of visual element (stack frame, environments, functions, and

data structures) is presented at least once.

7.4.1 The Surveyed Tools

The comparison includes the following tools:

Figure 7.14: A Screenshot of the Environment Model Visualization Tool



The Environment Model Visualization Tool (hereafter "the Env tool") Originally developed by Cai et al. [6] and later extended by Abad and Henz [1], this tool implements the environment model as presented in *Structure and Interpretation of Computer Programs* (SICP) [2]. A screenshot of the Env tool is shown in Figure 7.14.

Figure 7.15: A Screenshot of the Online Python Tutor Running Python

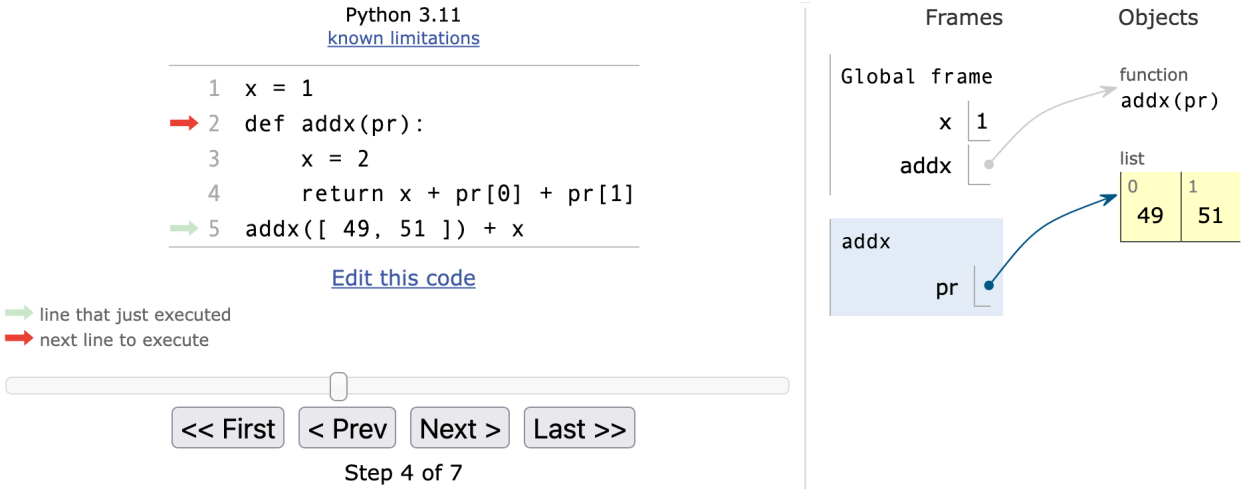


Figure 7.16: A Screenshot of the Online Python Tutor Running JavaScript

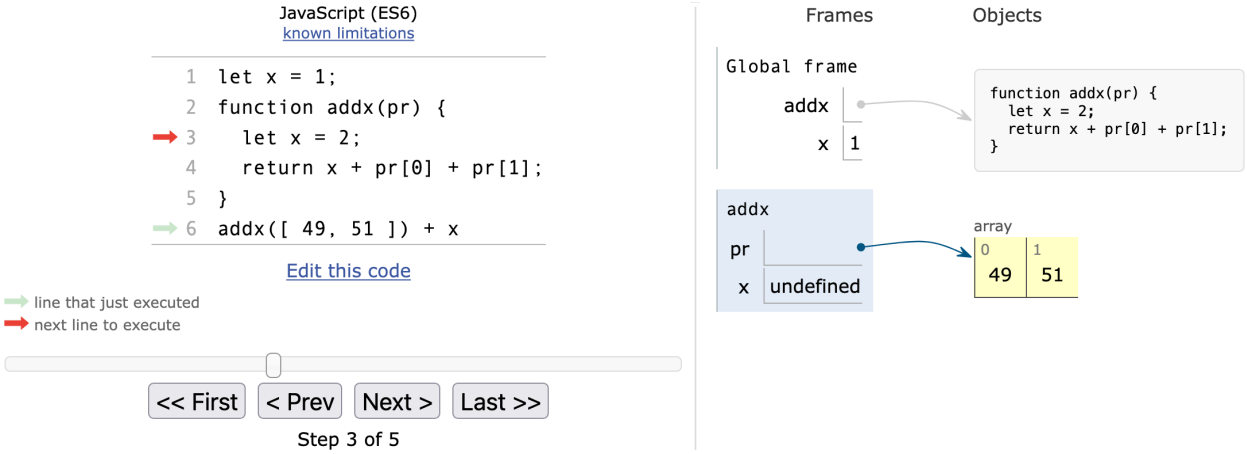
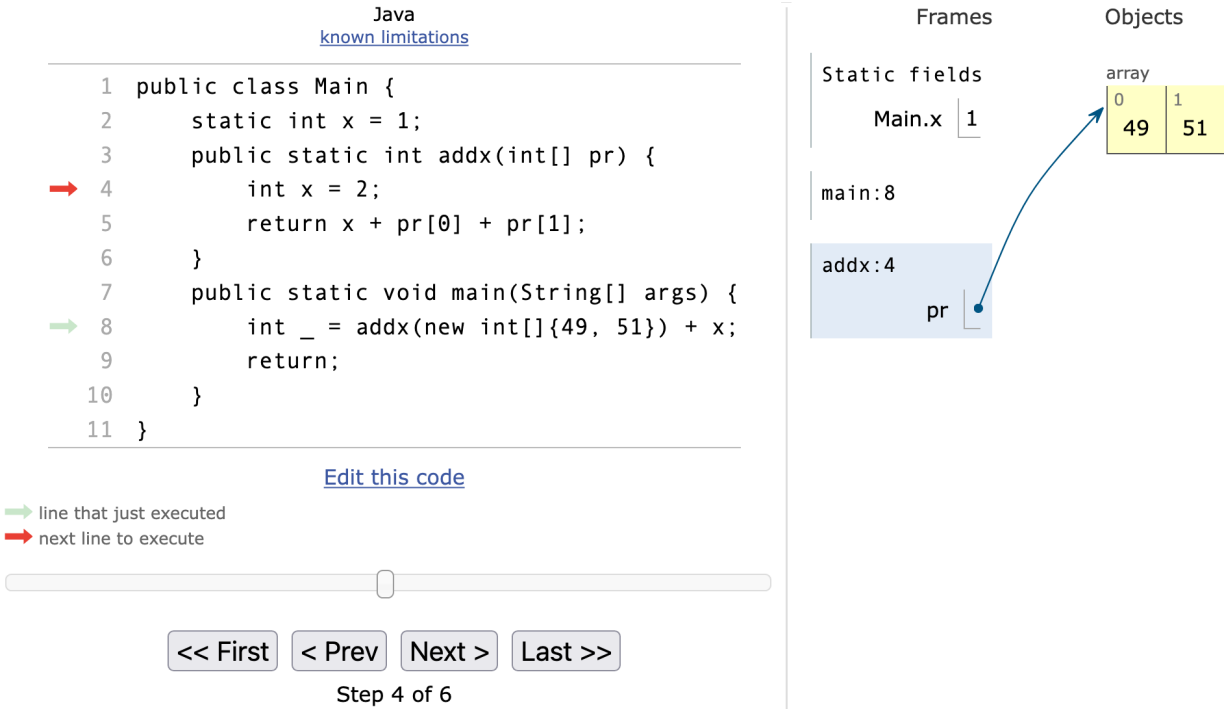
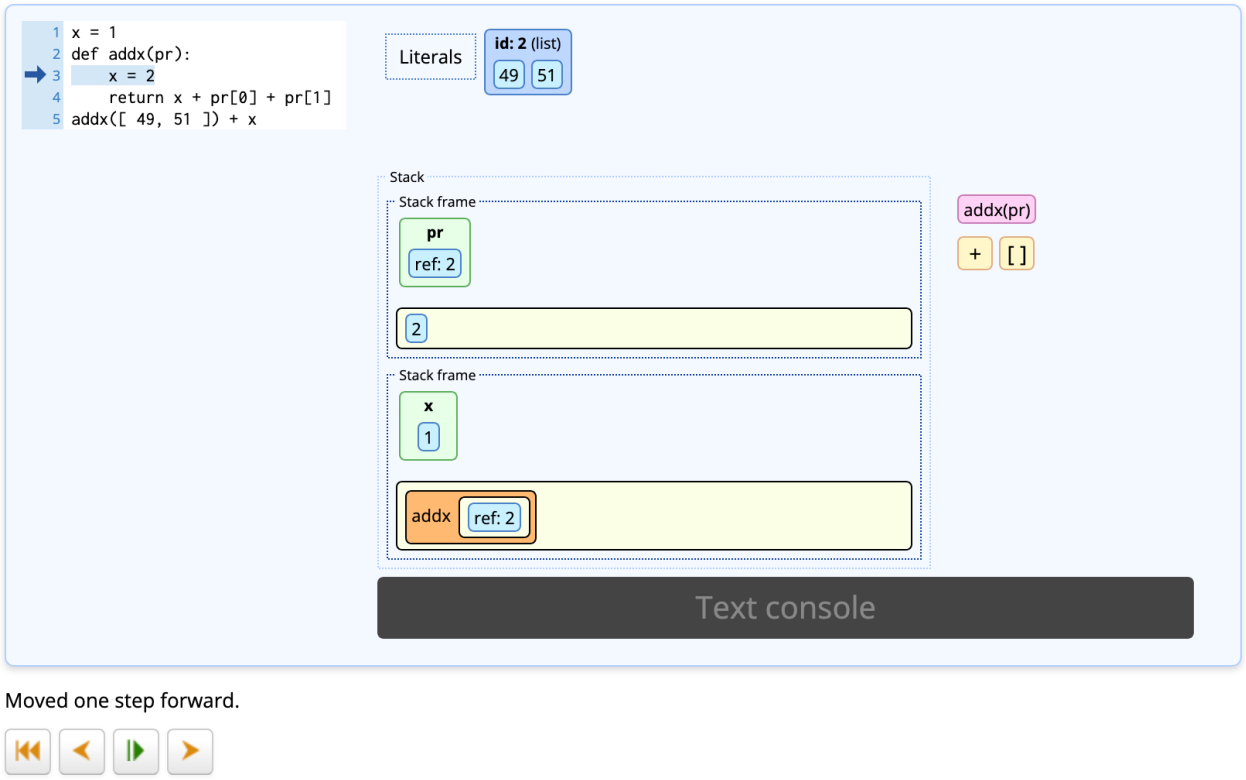


Figure 7.17: A Screenshot of the Online Python Tutor Running Java



Online Python Tutor (hereafter "OPT") Developed by Guo [25], this tool supports multiple language. Figures 7.15 to 7.17 show screenshots for different language configurations.

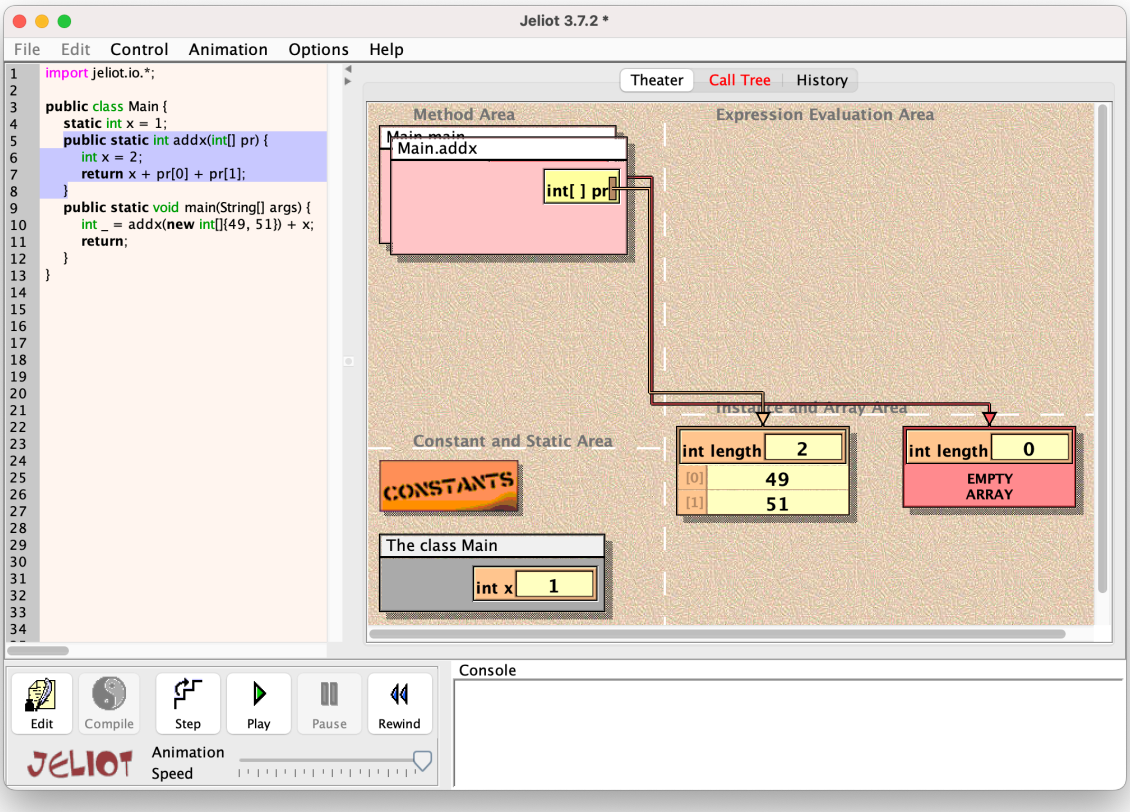
Figure 7.18: A Screenshot of the Jsvee & Kelmu



Jsvee & Kelmu (and its predecessor UUhistle) (hereafter "JK") According to UUhistle's official website (`http://www.uuhistle.org`), Jsvee & Kelmu [47] are successors of UUhistle [50]. Figure 7.18 shows a screenshot of JK.[1]

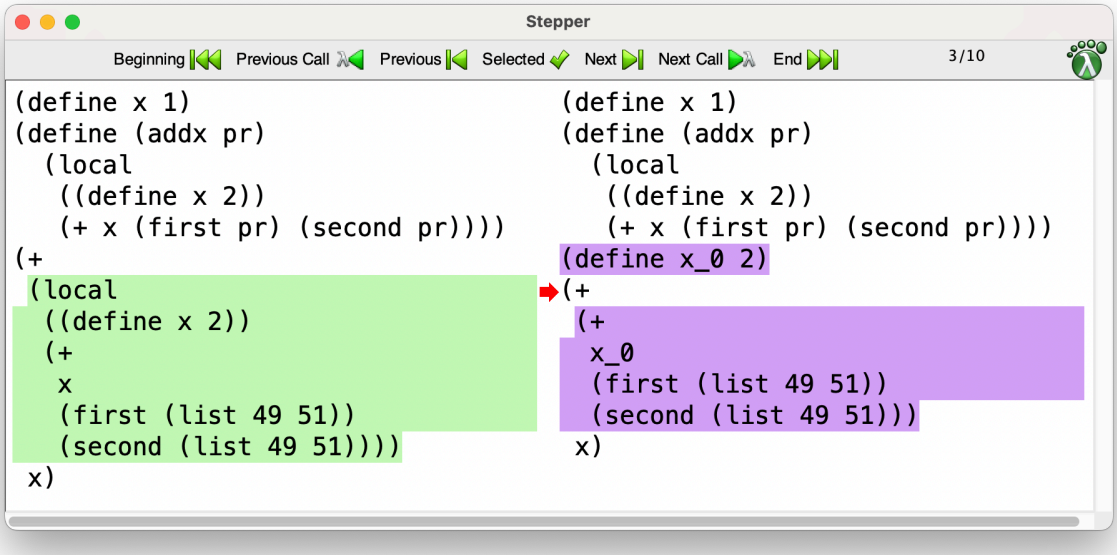
[1]Both systems include more functionalities than program trace visualization tools: UUhistle is the pioneer of *visual program simulation*, systems that support users to construct program states as a way to practice their understanding; Jsvee & Kelmu provides many features for *tailoring* program animations. In this work I focus on their usage as program trace visualization tools.

Figure 7.19: A Screenshot of Jeliot



Jeliot A series of visualization tools summarized by Ben-Ari et al. [5]. Figure 7.19 shows a screenshot of the latest version (Jeliot 3).

Figure 7.20: A Screenshot of Stepper



The Algebraic Stepper in DrRacket (previously known as DrScheme) (hereafter "Stepper") Developed by Clements, Flatt, and Felleisen [10]. Figure 7.20 shows a screenshot of Stepper.

There are many other similar tools. Given limited resources, I selected a subset of representative tools. Table 7.3 explains why they are selected. For a more comprehensive survey, see Sorva's dissertation [49].

7.4.2 Differences in Presented Information

Tables 7.4 to 7.7 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 7.4; See Section 7.7.1 for further discussion)

The Continuation What the system will do after the current task is complete. (Table 7.5;

Table 7.3: Basic Information about the Compared Tools

Tool	Why Included?	Source URL	Languages
Env	A recently developed tool.	`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 backed by 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.

[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, 32].

[d]See the Stepper documentation for more details.

See Section 7.7.2 for further discussion)

Environments How the tool represents the bindings and scopes of variables. (Table 7.6)

The Heap How the tool shows heap-allocated values, including functions and data structures. (Table 7.7)

Sections 7.7.1 to 7.7.4 further discuss the differences, including proposing alternative designs and comparing the pros and cons of the design options.

7.4.3 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.

Only OPT provides explicit visual feedback about the *previous* current task, using a light green arrow to point to the corresponding line in the source code.

Addresses vs. Arrows Environments and values form references. Only the Stacker and the JK present the references as textual labels. The Env tool (Figure 7.14), OPT, and Jeliot present references as arrows. (This discussion does not apply to the Stepper.)

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.

Only JK and Jeliot allow users to control the granularity of steps using a slider.

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

Table 7.4: A Comparison on the Presentation of the Current Task. See Section 7.7.1 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 7.5: A Comparison on the Presentation of the Continuation. See Section 7.7.2 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 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 7.6: A Comparison on the Presentation of Environments. See Section 7.7.3 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 7.7: A Comparison on the Presentation of the Heap. See Section 7.7.4 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.

Color Coding All tools except the Env tool (Figure 7.14) 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 7.14), 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 UUhistle present transitions between states using smooth animations.

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

During the early development of the Stacker, I conducted a study comparing it with the Stepper.

7.5.1 Population

The study was conducted in the Accelerated Intro course (see Chapter 4 for details).

Students in this course may have had some prior experience with debuggers and some familiarity with the Stepper, either from reading *How to Design Programs* (HtDP) or from interacting with DrRacket. However, they had never received formal instruction on using

Figure 7.21: The programs used in the Stacker-vs-Stepper study (Section 7.5)

```
;; 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 7.22: The special-version Stacker used the Stacker vs Stepper study (Section 7.5)

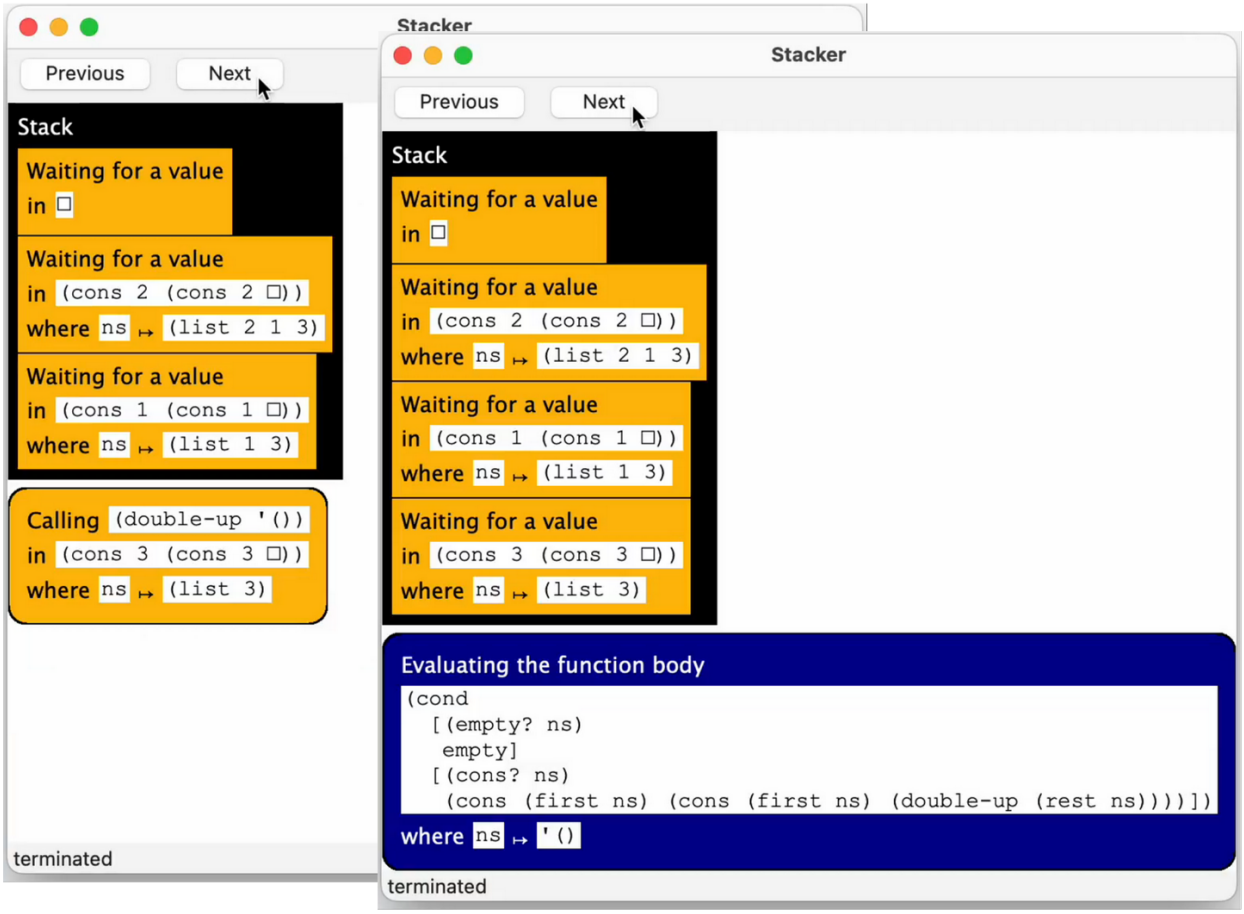
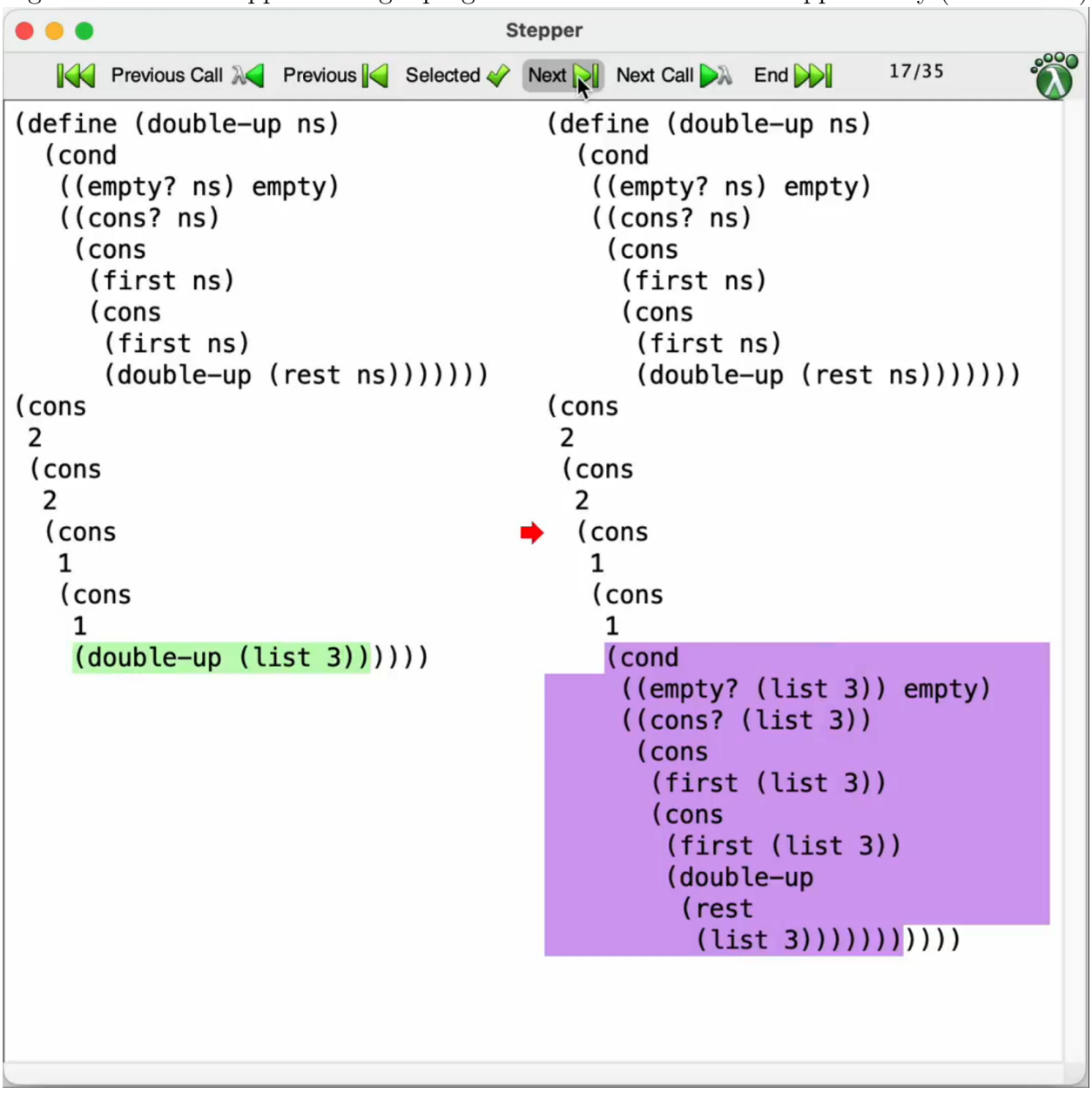


Figure 7.23: The Stepper tracing a program used the Stacker vs Stepper study (Section 7.5)



the Stepper. They had certainly never seen the Stacker, and likely had never encountered evaluation contexts.

7.5.2 Key Tool Differences

This study uses a special version of the Stacker (shown in Figure 7.22). 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 7.23 illustrates the Stepper showing the "same" state as in the Stacker figure (Figure 7.22).

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.

7.5.3 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.

7.5.4 Study Content

The study used two programs (Figure 7.21), 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.

7.5.5 Results

Perhaps unsurprisingly, 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 7.7.6 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 7.7.2 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 about 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 7.5.3 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.

7.6 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 4) in 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.

7.6.1 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 (Section 3.1):

```
(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?

7.6.2 Results and Discussion

Students noticed and liked the column-based design shared by both tools. See Section 7.7.7 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 7.7.7 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 7.7.1 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 7.7.14 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 7.7.8 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 7.7.3 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 7.7.3 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 7.7.5 for further discussion.

7.7 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 7.4) and new ideas inspired by the survey and the user studies (Sections 7.5 and 7.6).

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

Sorva's dissertation [49] 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.

7.7.1 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 7.6). Designers must choose between highlighting the *line*, as in the Env tool (Figure 7.14), OPT (Figures 7.15 to 7.17), and JK (Figure 7.18), or highlighting the *term*, as in Jeliot (Figure 7.19). 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 7.14) uses a distinct border color, while OPT (Figures 7.15 to 7.17) changes the background color. JV (Figure 7.18) always places the current environment within the top-most stack frame. Jeliot (Figure 7.19) 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 7.20) 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).

7.7.2 Presentation of the Continuation

Breaking the Continuation into Segments The Stepper (Figure 7.20) and the Env tool (Figure 7.14) 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 7.5). 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 7.13 and 7.15 to 7.19) 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 7.20) and the Stacker (Figure 7.13), or as a sequence of instructions, as in the Env tool (Figure 7.14). 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 7.15 to 7.17), JK (Figure 7.18), and Jeliot (Figure 7.19), integrate environments into stack frames. Others, including the Stacker (Figure 7.13) and the Env tool (Figure 7.14), present environments and stack frames as clearly separate entities.

Clements and Krishnamurthi [11] 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 7.13) or partially (as in JK, Figure 7.18);
- 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 7.3, 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 7.5).

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.)

7.7.3 Presentation of Environments

Substitution vs. Environments Substitution, as used in the Stepper (Figure 7.20), 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 6.1).
- Substitution lacks authenticity with respect to real-world language implementations.

Compared to substitution, environments are more helpful for tracking function calls (Section 7.5).

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 7.13) and the Env tool (Figure 7.14) 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 7.13), the Env tool (Figure 7.14), and OPT with JavaScript (Figure 7.16). Showing these variables is critical for understanding hoisting behavior, which is present in many modern languages (e.g., Python and JavaScript, but not Java).

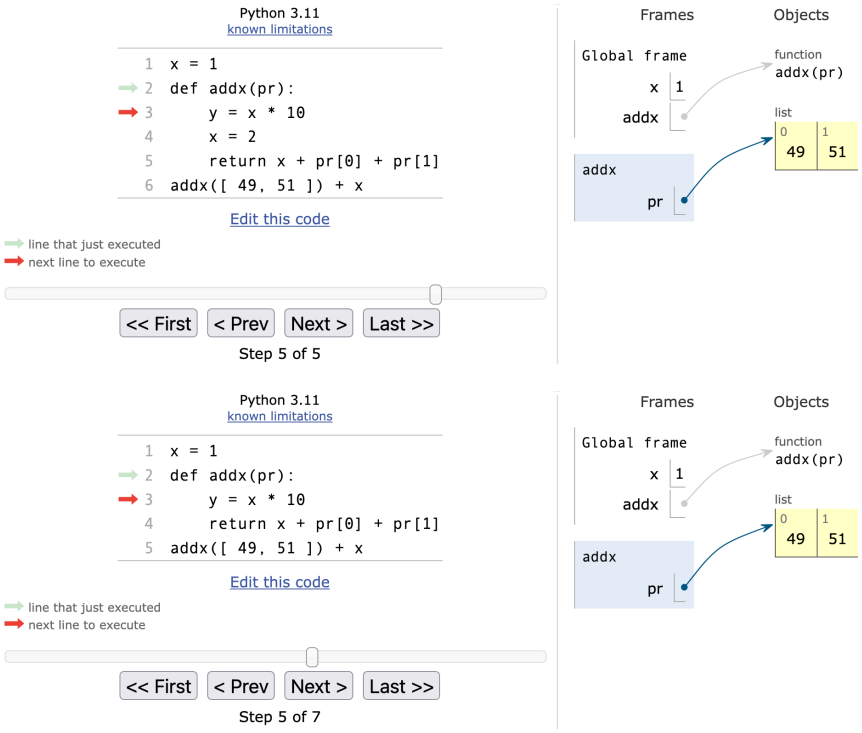
Consider the following two variants of the same program (used in Section 7.1 and Section 7.4). In the first, the line `y = x * 10` appears *before* `x = 2`:

```
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 7.6).

The Environment Hierarchy Some surveyed tools, including the Stacker (Figure 7.13), the Env tool (Figure 7.14), and Jeliot (Figure 7.19), always make the environment hierarchy explicit. Others, like OPT with Python (Figure 7.15) and JavaScript (Figure 7.16), highlight only the top-level environment. Some tools, like JK (Figure 7.18), 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 7.13)—like all other surveyed tools (Figures 7.14 to 7.19) but unlike its predecessor [11]—does not distinguish between parameters and locally defined variables.

In my experience, making this distinction has not been helpful. Unless the tool targets a language where this distinction is meaningful, it is probably better not to separate the two.

7.7.4 Presentation of the Heap

Separating Different Kinds of Heap-allocated Values JK (Figure 7.18) 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 7.19) 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 7.13 to 7.16) follow this design.

Color-coding vs. Shape-coding The Env tool (Figure 7.14) uses different shapes to distinguish function values and data structures. The Stacker (Figure 7.13) and JK (Figure 7.18) use the same shape but different colors. OPT (Figures 7.15 to 7.17) 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 7.15) and JK (Figure 7.18), and the source code location, shown by the Stacker (Figure 7.13), 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 7.7). 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 4) 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 7.13), the Env tool (Figure 7.14), and Jeliot (Figure 7.19)[2], but not in OPT (Figures 7.15 to 7.17) or JK (Figure 7.18). In JK's case, this might be reasonable since all environments are top-level due to its language restrictions (Table 7.3). 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 7.13) 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. 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

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

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 7.14), OPT (Figures 7.15 to 7.17), and Jeliot (Figure 7.19). I think it's helpful to include indices. (**Future Work:** Present vector indices in the Stacker.)

7.7.5 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 7.6), 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 7.14) hides most details about function values unless users hover over them (Table 7.7). The Stacker reveals those details only when users click on the function's source location (Section 7.1).

7.7.6 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 7.1.2 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 [55, 28, 20].

Similarly, states before print statements are also retained, despite not triggering major visual changes, because prior work Lu et al. [31] has shown that print behavior can be confusing.

(**Future Work:** Having identified new misconceptions (see Chapter 6), 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 7.4. 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 7.5), 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 7.20) 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.)

7.7.7 Major Areas

Arrangement of Areas The Stacker (Figure 7.13), the Env tool (Figure 7.14), OPT (Figures 7.15 to 7.17), and the Stepper (Figure 7.20) organize their visual elements into columns. JK (Figure 7.18) and Jeliot (Figure 7.19) 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 7.6), and it aligns with the reading order of many natural languages.

Area Labels The Stacker (Figure 7.13), OPT (Figures 7.15 to 7.17), and Jeliot (Figure 7.19) label all their major areas. The Env tool (Figure 7.14) does not label any. I believe tools should include area labels—students appreciate them (Section 7.6), and there are no clear downsides.

7.7.8 Textual vs. Arrow References

References are represented textually in the Stacker (Figure 7.13) and JK (Figure 7.18), and as arrows in the Env tool (Figure 7.14), OPT (Figures 7.15 to 7.17), and Jeliot (Figure 7.19).

Textual references necessarily occupy more screen space: each arrow must be replaced with one or more pieces of text, which might be shared across multiple references. 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 7.6).

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, textual labels require users to search across the screen to match identifiers.

Figure 7.24: Online Python Tutor presenting a cyclic data structure

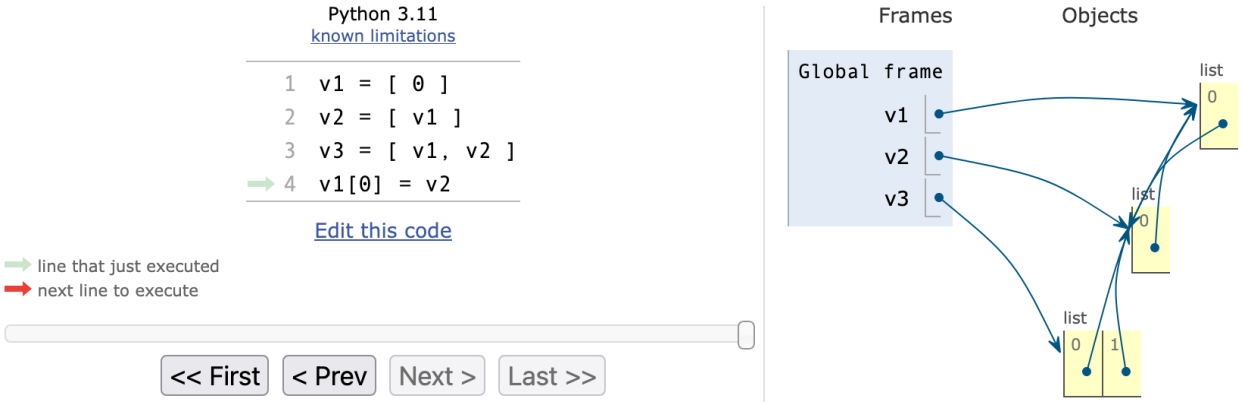
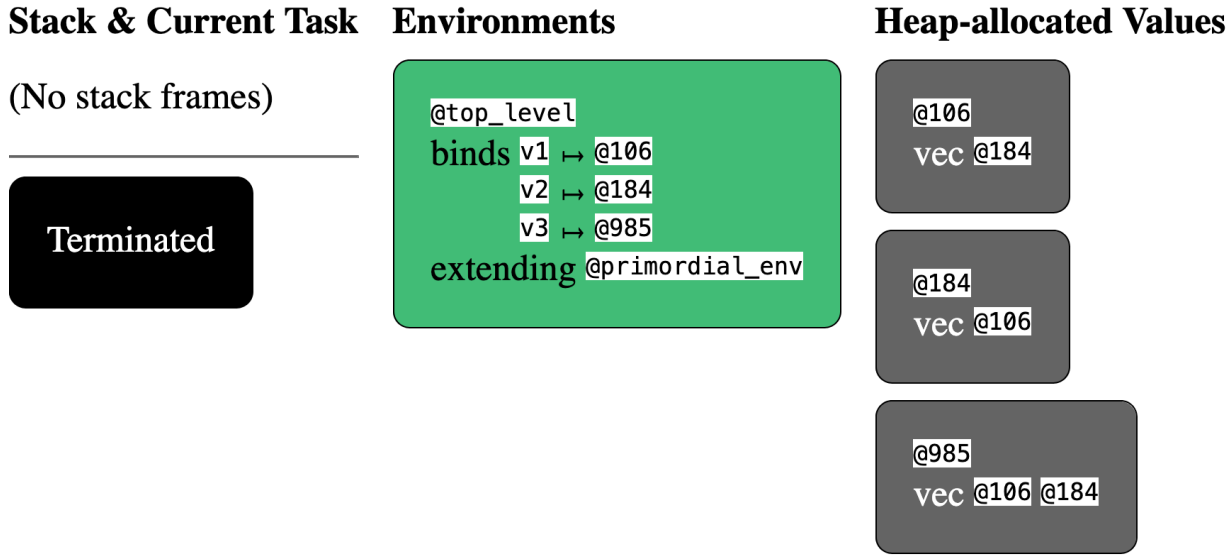


Figure 7.25: Stacker presenting a cyclic data structure



However, the usability of arrows is highly sensitive to layout. Aesthetic aspects such as symmetry, edge crossings, and bends greatly impact graph readability [39]. Achieving a clean layout is non-trivial. For example, Figures 7.24 and 7.25 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 textual references are less sensitive than arrows to reference complexity.

Textual references are also more authentic (with respect to real programming language implementations), especially when they resemble memory addresses.

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 Display arrows only when the address is hovered over—reducing clutter while preserving clarity.

The Stacker originally avoided arrows because they were harder to implement on the web and because textual authenticity was seen as valuable in the PL course where it was first deployed. After the Stacker-vs-OPT study (Section 7.6), I implemented the "Textual + Highlighting" approach in response to student feedback (Section 7.2.8). (**Future Work:** Implement the "Show Arrows on Hover" approach as well.)

7.7.9 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 incredibly helpful in education.

Web-based tools differ in their server cost models. OPT appears to be the most expensive, based on its advertisements and donation prompts. In contrast, the Stacker operates with essentially zero server cost:

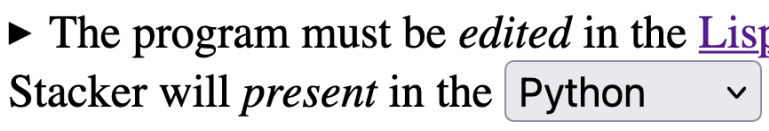
- The only cost is serving the static page load, which is cacheable via CDNs.
- Evaluation happens entirely in the browser.
- Permalinks are self-contained `GET` requests, requiring no server-side state.[3]
- There is no persistent server state, so there are no ongoing space or time costs.

7.7.10 Supported Language(s)

Supported Constructs I recommend supporting at least all constructs in SMoL, as they are ubiquitous in modern programming languages.

Tool designers should also consider supporting looping constructs, which are similarly widespread, as well as generators and async functions, which are becoming increasingly important. (**Future Work:** Add these constructs to SMoL and to the Stacker.)

Support 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:



[3] `GET` has size limits, which bound the size of programs that can be shared. In practice, this has not been a problem.

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.)

7.7.11 Smooth Transitions Between States

State transitions are animated in JK (Figure 7.18) and Jeliot (Figure 7.19). Jeliot even provides a slider to control animation speed. Smooth transitions help users follow state updates, though they may be challenging to implement.

7.7.12 Accessibility Concerns

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

The color palette (Table 7.1) is designed with accessibility in mind:

- Background colors were verified using Adobe Color [13] to be colorblind-friendly.
- Most foreground-background combinations meet WCAG 2.0 level AAA contrast standards, verified via WebAIM [59]. The only exception is white on #646464, which meets level AA.

7.7.13 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.

7.7.14 Trace Navigation

First & Last Buttons

In the Stacker-vs-OPT study (Section 7.6), students expressed appreciation for OPT's "First" and "Last" buttons (Figures 7.15 to 7.17), which jump to the start and end of the trace.

A "First" button is also present in the Env tool (Figure 7.14), OPT, and JK (Figure 7.18). 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 7.14), OPT (Figures 7.15 to 7.17), and the Stepper (Figure 7.20). 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 7.6) 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.

CHAPTER 8

SMOL TUTOR

This chapter introduces SMoL Tutor, an interactive, self-paced system designed to address SMoL misconceptions. Section 8.1 provides a guided tour. Section 8.4 evaluates the tutor, demonstrating its effectiveness in correcting certain misconceptions. Section 8.7 explores the design space.

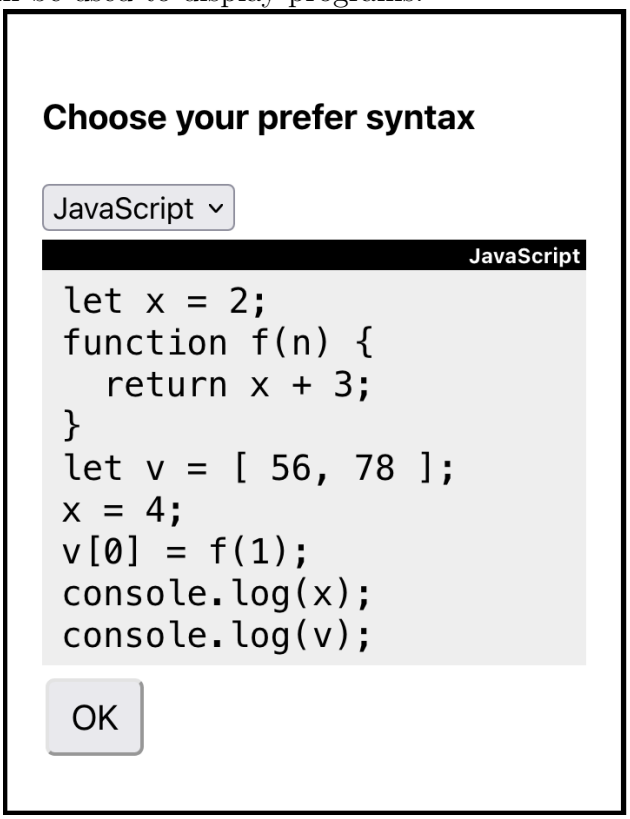
8.1 A Guided Tour of SMoL Tutor

8.1.1 Choose a Syntax

When users open a Tutor, they first encounter a dialog prompting them to select their preferred syntax (Figure 8.1). The dropdown menu lists syntaxes supported by the SMoL Translator (Section 3.2). Changing the syntax updates the example program accordingly. The tutor's behavior varies based on the instructor's configuration:

- If the instructor *suggests* a syntax, it appears as the default option, but students can still select a different one.

Figure 8.1: Users can select their preferred syntax when they first open the SMoL Tutor. The selected syntax will be used to display programs.



- If the instructor *sets* a syntax, it is preselected, and the dialog does not appear for students.

8.1.2 Choose 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 8.1. 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.

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").

8.1.3 Interpreting Tasks

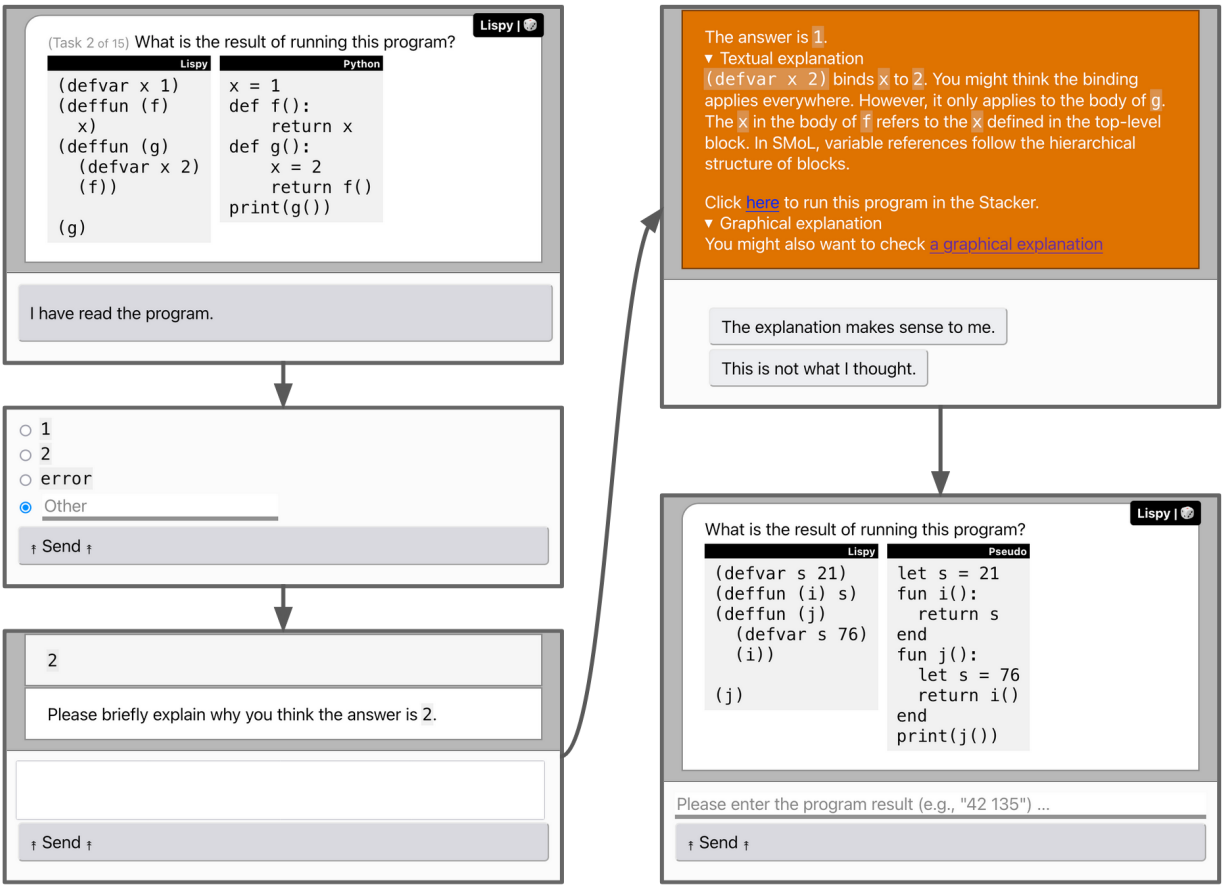
Each tutorial consists of a sequence of tasks, most of which are **interpreting tasks**. These tasks (illustrated in Figure 8.2) begin with multiple-choice questions (MCQs) designed to detect misconceptions. Students may also be asked to justify their choices, as shown in the figure. After selecting 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 follow-up question.

Each interpreting task has one role in the Tutor. It can be either a **warm-up** task, a

Table 8.1: 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 compound data. • 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.

Figure 8.2: An Interpreting Task in the SMoL Tutor.



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* (Chapter 9), followed by a link to open a Stacker trace. Some explanations also include a link to a specialized version of 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.

Each program is displayed in two syntaxes. The left side shows the primary syntax selected through the process described in Section 8.1.1. The right side displays a randomly chosen syntax, which users can change by clicking the black dice button.

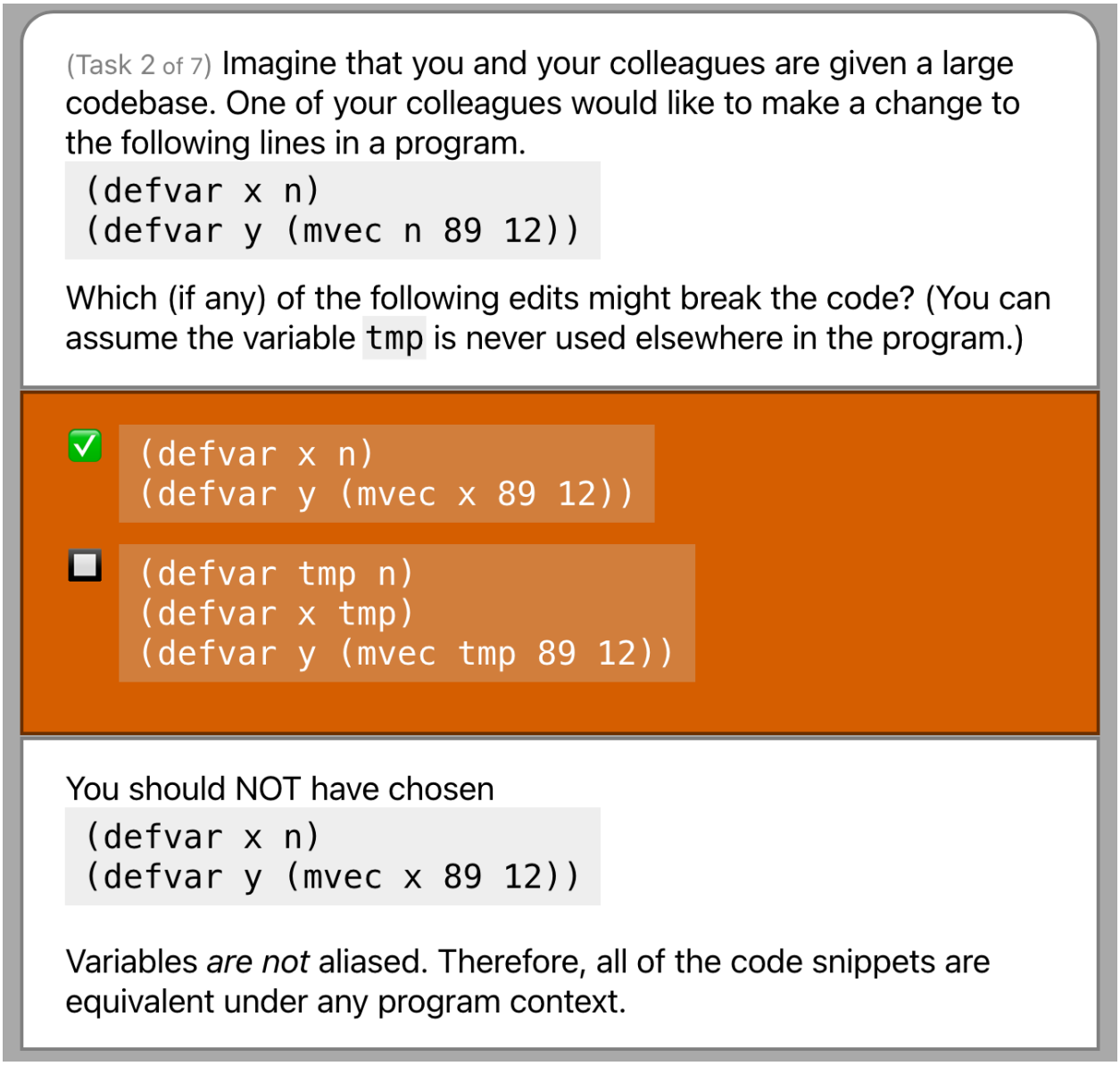
Consecutive interpreting tasks within each tutorial are presented in a randomized order.

8.1.4 Equivalence Tasks

Interpreting tasks ask students to evaluate programs. Although similar tasks are commonly used to detect misconception, they lack authenticity (Section 6.3). In real programming scenario, people rarely run programs in minds. In contrast, it is not uncommon for programmers to compare two versions of program fragments in refactoring and code review, etc. Equivalence tasks aim to simulate these situations.

Figure 8.3 illustrates an answered equivalence task. The task first presents an *original* program fragment and a collection of *alternative* fragments, and then asks students to select fragments that "*might break the code*". In the example, none of the alternative fragments might break the code, with the given assumption that the `tmp` variable is not otherwise used.

Figure 8.3: An Equivalence Task in the SMoL Tutor.



So checking the first option is wrong, as indicated by the orange background. In this case, the Tutor provides negative feedback, which first lists what should (not) be chosen, and then justify the correct answer.

There are six equivalence tasks, each target one aliasing misconception (Table 6.1). They are presented in a random order in the **refactor** tutorial.

8.1.5 Other Components in 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.
- Tutorials present their learning objectives to students as conclusions of the tutorials.
- After completing a tutorial, students are encouraged to download a PDF copy for review. This feature was added based on requests from early users who wanted a way to revisit the material without repeating the entire tutorial.

The Tutor also includes special tasks beyond interpreting tasks (Section 8.1.3) and equivalence tasks (Section 8.1.4):

- 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.

8.2 SMoL Tutor Versions and Studies

Multiple studies have been conducted with different versions of the SMoL Tutor. Some versions introduced significant updates (e.g., substantial changes in question sets and randomization of question order) that make it difficult to compare studies across versions. Therefore, this section first introduces the major versions and then list studies related to each version.

The SMoL Tutor experiences a major update roughly every half year. I name major versions as 'Mid 2022', 'Early 2023', 'Mid 2023', etc. Minor updates, such as bug fixes, English improvements, and minor tweak in UI and question programs, are made throughout the year. I have no reason to believe that the minor updates have significant impact on the overall Tutor experience. Therefore, I will ignore the minor updates and focus on the major versions.

The first version is Mid 2022. The Early 2023 version made many updates, but they are all incremental or fixing now obvious problems in the initial prototype. Each of the later versions made significant updates:

Mid 2023 introduced language switch. Previously, the Tutor present programs only in the SMoL syntax. This version also introduced web-based stacker, with which students can trace programs by clicking a "run" button. Previously, to trace a program, students have to copy the program, paste it in DrRacket, and hit the run button.

Early 2024 made substantial changes to the question sets in light of misinterpreters. This version also randomized question order, introduced experimental support for Scala 3, and introduced the two language display for programs. Previously, the Tutor displays programs in only one language. Students can switch the language by clicking a button.

Mid 2024 introduced equivalence tasks (Section 8.1.4)

There isn't an **Early 2025** version because my personal scheduling constraint.

This dissertation focuses on studies conducted with the two latest version ('Early 2024' and 'Mid 2024'). All population names are defined in Chapter 4.

The Mid 2024 Tutor was delivered to the PL course. In that iteration of the course, 46 students were enrolled. All of them completed the Tutor.

The Mid 2024 Tutor was also delivered to the Belgium University. 126 students completed the Tutor.

The Early 2024 Tutor was delivered to the University 2, where 13 students used the Tutor, and the Sweden University, where 56 students used the Tutor.

8.3 Evaluation: Coverage of Named Misconception

This section concerns with the following questions: In the interpreting tasks, how many of the wrong answers are associated with the named misconceptions? What are the pattern (if any) of the remaining wrong answers?

8.3.1 Most wrong answers are associated with a named misconception.

Table 8.2: Distribution of Answers to Interpreting Tasks by the Conception Kind. Percentages might not sum to 100% due to rounding error.

	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%

Table 8.2 provides a high-level view of the distribution of answers by their associated (mis)conceptions. Students provide the correct answer over half of the time. In most institutions (with the US2 being the exception), 3/4 to 4/5 of the wrong answers are associated with a named misconception. Readers probably should not read too much into the numbers about US2 because that population is much smaller than the other populations (13 vs 46-

126). A Fisher's exact test confirms that the difference between US2 and non-US2 is not significant ($p = 0.3329$).

8.3.2 Wrong answers not associated with named misconceptions

Those wrong answers suggest new misconception. Therefore, we should check the associated program and students' justification for their answers. I present a selected subset of the wrong answers because there is a long tail of those. The subset is selected by the following process:

1. Compute the frequencies of wrong answers.
2. If the highest frequency is more than 150% of the second-highest different frequency, I include the wrong answers corresponding to the highest frequency in my analysis and then go back to the first step. Otherwise, I stop.

This process is done individually with each population. The results, however, are presented together because a wrong answer common to one population is often also common to another population.

For each of the presented wrong answer, all collected justifications are presented. There are very few of those because students are not always asked to justify their answers, as explained in Section 8.1.3.

Some students selected `3` as the output of the following program

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

The following justifications are collected:

- *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 seems to choose the wrong answer by mistake. The second student correctly note that the top-level `y` is not bound, but nevertheless selected `3` rather than the correct answer `error`. I think this student might believe unbound variables are resolved to `0` or ignored.

Some students selected `error` as the output of the following program:

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

One justification is collected: *x in f undefined*. It is unclear to me that why this student think `x` is not bound: The student might overlook the first statement, or believe that functions can not refer to variables defined in the top-level block.

Some students selected `error` as the output of the following program:

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

Two justifications were collected:

- *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.

Some students selected `58` as the output of the following program:

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

The wrong answer treats 1 as 0 in vector indexing. The Tutor did not collect any justification because this is a warm-up question (Section 8.1.3).

Some students selected `67 73` (whereas the correct answer is `#(67 73)`, which include brackets around the vector) as the output of the following program:

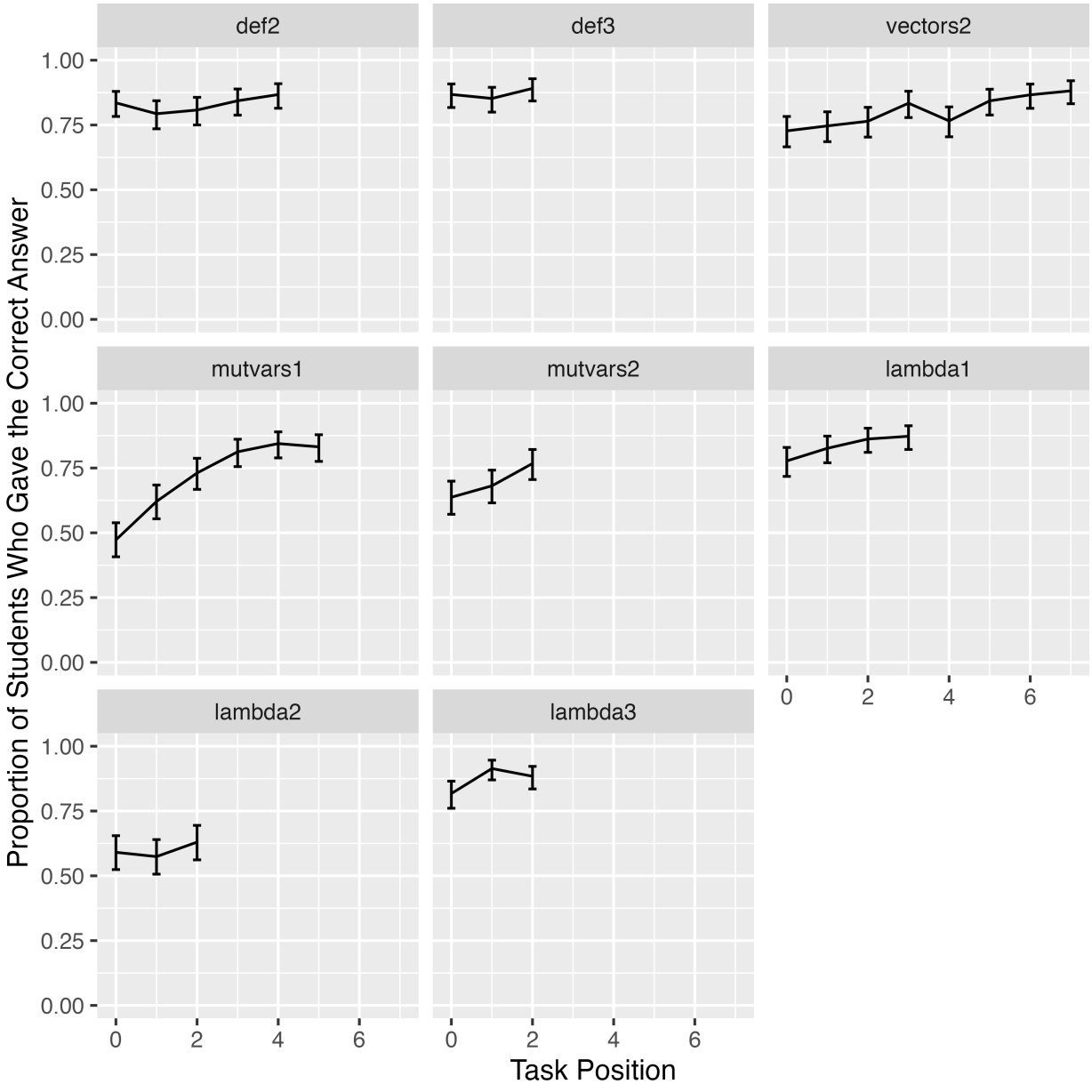
```
(defvar x (mvec 92 73))
(vec-set! x 0 67)
x
```

The Tutor did not collect any justification because this is a warm-up question (Section 8.1.3).

8.4 Evaluation: How Effective is SMoL Tutor?

This section answer the question from two perspectives. If the Tutor is effect, as students do more practice with the Tutor, they should be more likely to give the correct answer and, for each misconception, more likely to avoid the corresponding incorrect answers. Technically speaking, we only need the first perspective to tell if the Tutor is effective. But the second perspective is particular useful for deeper insights:

Figure 8.4: Students seem to be more likely to give the correct answer in later tasks.



- I hope the Tutor is effect mainly by correcting misconceptions. It is possible that students improve on giving the correct answer but do not improve on avoiding misconception-related wrong answers by selecting less "random" incorrect answers. If this happens, the Tutor is not effective in the expected way. We can't rule out this possibility without another perspective.

- The design of each Tutor task highly depends on the underlying misconceptions. The second perspective provides more details about each misconception, and hence is more useful for knowing the design space of the Tutor. For instance, if we find that the Tutor is usually effective on all but a couple of misconceptions, we know that there might be something wrong with the misconception-related programs or the refutation text.

All analysis in this section lump the four populations together.

8.4.1 Converge to the correct answers

Figure 8.4 shows how the proportion of students who gave the correct answer changes as students do more tasks. It suggests that students improve. However, the improvement might or might not be clear in some tutorials. Therefore, I do a statistical analysis to check the trends.

A straightforward statistical analysis would perform a logistic regression between position of tasks and the correctness. However, there are a few additional considerations:

1. As students complete more related tasks, they also become more familiar with the UI. We need to verify that improvements are due to conceptual learning rather than simply learning how to interact with the system.

2. The difficulty of interpreting tasks varies, and the randomization of question order may not be perfectly balanced. If tasks happen to be ordered in a particular way (e.g.,

if difficult tasks are disproportionately placed at the beginning for most students), it could create a false 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 8.4. However, I don't see such trend in the plot. In fact, logistic regression shows that the trend is actually downward with an effect size -0.012 ($p < 0.001$).

Table 8.3: 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

The second possibility, however, remains a concern. To address this, I perform a model selection to determine which of the following models is most plausible for each tutorial:

Task-Specific The proportions depend only on the tasks. This model represents the situation of concern.

Position-Specific The proportions depend only on the position. If this model is most plausible, the tasks have similar difficulty and students are more likely to give the correct answer as they do more tasks.

Both Task and Position Matter If this model is most plausible, the tasks vary in their difficulty and students are more likely to give the correct answer as they do more tasks.

To find the most plausible model, I 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 8.3 presents the AICs and the effect of position in the better one of the position-included models. As expected, the Task-Specific model is never the best. In five out of the eight applicable tutorials, the effect of position on students' performance is clearly ($p < 0.05$) positive.

The DEF2 and DEF3 tutorials do not have a clear positive effect probably due to ceiling effect: Students' performance is already high when they started the tutorials (Figure 8.4).

The lack of clear improvement in LAMBDA2 is perhaps interesting. The tutorial is the first tutorial that shows higher-order functions with interesting interactions with scope. Its programs are designed to correct the FLATENV and DEEPCLOSURE misconceptions in the context of first-class functions. These topics are, at least anecdotally, difficult. So a three-task tutorial is probably too short. Given that the trend is suggestively good (Table 8.3), the Tutor should probably add more tasks to the tutorial.

8.4.2 Avoid misconception-related incorrect answers

This section evaluates the effectiveness of the Tutor from a second perspective: If the Tutor is effective, for each misconception, students should be less likely to choose answers associated with the misconception as they complete more related tasks.

Figure 8.5 shows, for each applicable combination of misconception and tutorial, how the proportion of students who avoided the misconception changes as students do more related tasks.

The analysis includes a model selection akin to the one in Section 8.4.1. I am looking for a negative (rather than positive) and **clear** effect, meaning that students improve as they progress through tasks.

Figure 8.5: Students seem to be more likely to avoid the related wrong answer in later tasks.

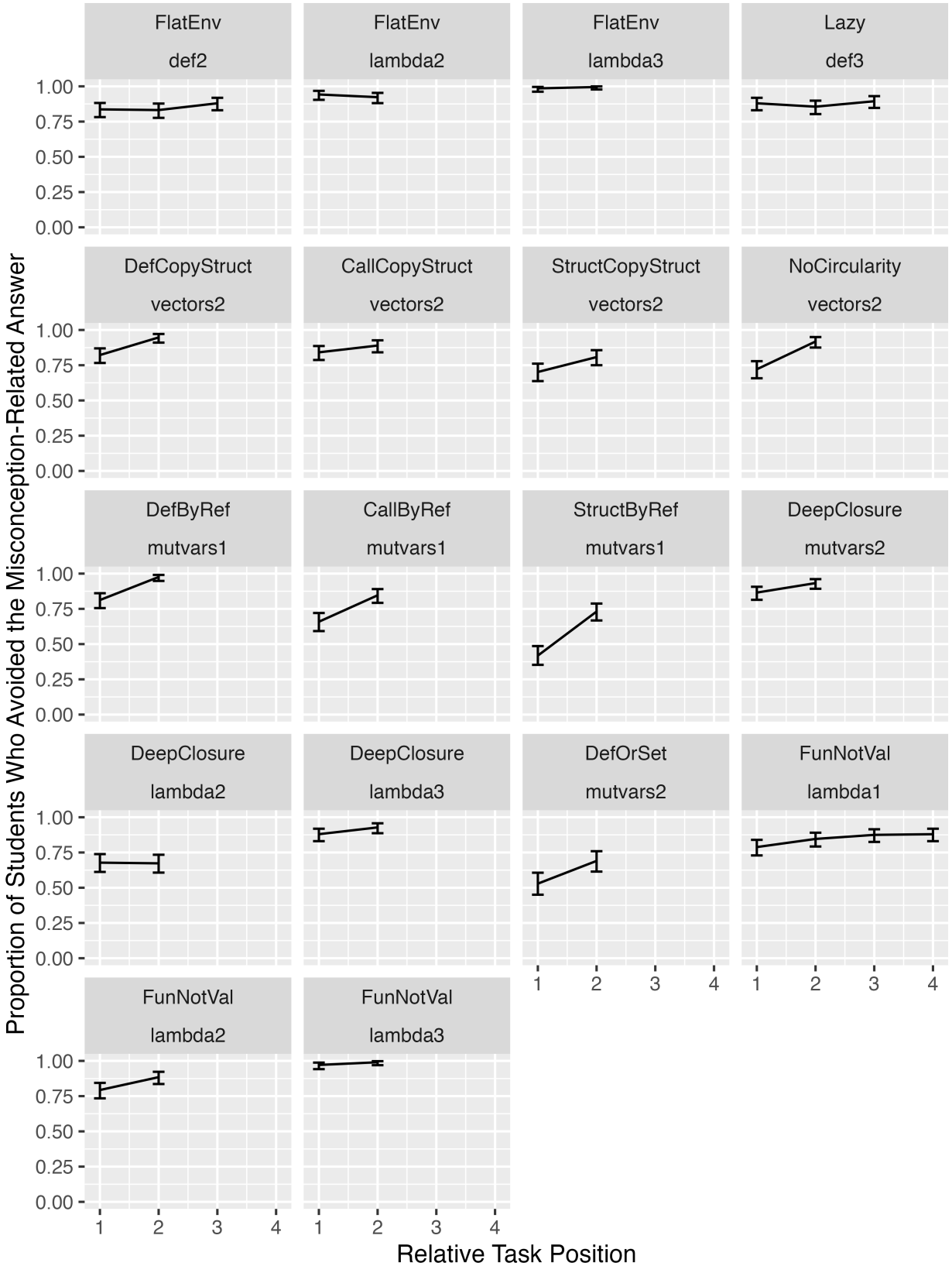


Table 8.4: Are students avoiding the same kind of mistakes?

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	$= 0.195$

Table 8.4 presents the results. As expected, the Task-Specific model is never the best. When the effect of position is clear, the effect is always positive.

The vague trends in FlatEnv-lambda3 and FunNotVal-lambda3 are probably artifact of the ceiling effect: students were already good at avoiding those misconceptions when they were first tested (Figure 8.5).

Ceiling effect can not explain the remaining bad trends. Further investigation is required to figure out how the Tutor can be more effective on correcting those misconception-topic combinations.

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

As illustrated in Figure 8.2, 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 8.4.2). 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 receives 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 8.4.2, 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 can not make sense out of the other responses.

8.6 Evaluate the Learning Objectives as Rules of Program Behavior

Rules of program behavior (RPBs) are "written statements about how computer programs of a particular kind behave when executed", and they specify learning objectives [17].

This section evaluates the SMoL Tutor's learning objectives (Appendix C) and SMoL Characteristics as RPBs according to the 16 evaluation criteria listed by Duran, Sorva, and Seppälä [17].

Because the learning objectives are essentially a refined version of SMoL Characteristics, the discussion should be considered applicable to both unless I stated explicitly.

Cultural Fit A perhaps unusual thing about my rules is that they are not designed to for people with little prior programming background; rather, it assumes the user has some basic facility with programming (i.e., that they are already familiar with concepts like vectors, mutation, and higher-order functions).

Accuracy Being multilingual, my rules can not be 100% accurate due to the diversity of languages. However, each of them is correct for many programming languages. The learning objectives also sacrifice accuracy for ease of education in the **def** tutorials, where they define recursive look-up mechanism of lexical scope directly in terms of blocks rather than going through environments. The learning objectives of the **mutvars**, a later module that introduces mutable variables, define environments and revisit the rules about lexical

scope. When there is no mutation, we don't really need "environments" to explain program behavior. If I introduce environments in the earlier tutorials, I cannot properly justify the introduction of the concept.

Coverage My rules cover mutable variables, mutable vectors, first-class functions, and their interactions.

Simplicity The "simplicity" in [17] is about the number of concepts and the complexity of dependencies between the concepts. It is not very interesting to evaluate SMoL Characteristics by this criterion: they are simple but probably too simple to provide sufficient details. The learning objectives are designed with simplicity in mind. The SMoL Tutor intentionally presents a mutation-free subset of the language in early tutorials (**def**) and introduce more concepts only when it needed: "heap" and "heap addresses" are introduced only after mutable vectors are introduced; "environments" is introduced only after mutable variables are introduced.

Consistency The consistency of my rules is compromised every time a language behavior different from the rules is documented. (If it is not documented, accuracy is compromised). My rules are explicit that eager and sequential evaluation holds only with the assumption that we are within a thread, and that aliasing mutable values and *not* aliasing mutable variables are not always true.

Granularity The learning objectives aim to be fine enough so that learners can evaluate every program covered by SMoL mechanically. The SMoL Characteristics is coarser.

Abstraction The learning objectives are designed to be abstract enough to avoid implementation details. The SMoL Characteristics are designed to be abstract enough to include many modern programming languages.

Expressibility Duran, Sorva, and Seppälä [17] defines "expressibility" as "how feasible it is to construct a student-facing form of the RPBs that is suitable for the target audience". By this criterion, the learning objectives are high in expressibility because the SMoL Tutor presents them as they are to students. Although I believe this is "suitable", it is unclear to me on how to evaluate this point.

Notational Fit My understanding of notational fit, given the examples in [17], is to what extent the program behavior is explained in terms of syntactic concepts. Explaining scope in terms of blocks is high in notation fit, but compromises **accuracy** as I have discussed. The introduction of "the heap" and "heap addresses" certainly reduce notional fit, but they are necessary to explain the behavior of mutable values correctly.

Generality My rules are designed to have high generality in the sense that they are mostly consistent across many languages.

Transferability This criterion seems to be designed for RPBs that target one language. So it is likely not applicable to my rules.

Sensitivity to Conceptions Duran, Sorva, and Seppälä [17] states that *"this criterion is concerned with how RPBs attend to the learners' expected prior knowledge, common (mis)conceptions, and difficult-to-learn content."* My rules expect prior knowledge about general programming but do not expect system-level knowledge. The rules themselves are not designed with misconceptions in mind, however, I am always able to draw contrast between the learning objectives and misconceptions when I develop refutation text for the Tutor. My rules attend to difficult-to-learn content in the sense that rules about first-class functions, which I believe is a difficult topic, are presented last, by when students would already have a more solid understanding about other concepts.

Other criteria It is unclear to me how to evaluate my rules on **Wieldiness**, **Assessability**, **Implementability**, and **Clarity of Writing**

8.7 The Design Space Around SMoL Tutor

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

8.7.1 SMoL Tutor UI Updates

Over time, the SMoL Tutor's UI has undergone several updates. This section highlights the major changes and their motivations.

Dual-Syntax Display Earlier versions presented programs only in the SMoL Language. The current version always displays programs in a fixed primary language alongside a randomly selected secondary language. This change aligns with the reasoning provided in Chapter 3.

Justification Requirement The older version did not require students to justify their answers. The current version does, allowing for deeper insight into whether students genuinely hold the anticipated misconceptions.

Randomized Task Order An older version does not randomize question order. The current version does. The benefit of fixed order is that we have better control over the learning curve. In the older version, I placed tasks that I believe are easier closer to the beginning of each tutorial. With this setting, however, there is no way to tell if a task is indeed easier from the data. In fact, fixed order is unfriendly to many statistical analysis, including my current evaluation (Section 8.4).

8.7.2 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, which I don't see clear downsides, and fully randomize the order of other interpreting tasks, which unfortunately sacrifices education value for easier data analysis.

Students only received feedback after having completed a whole quiz. At the end of each quiz, they received both summative feedback *and* a documents 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.

Students were also encouraged to run the programs, but I have little reason to believe that they did (and certainly they asked few questions on the class forum about them). The SMoL Tutor, in contrast, provides links for students to run programs in the Stacker and logs those events. Figure 8.6 shows how often students run programs in the Tutor. Some

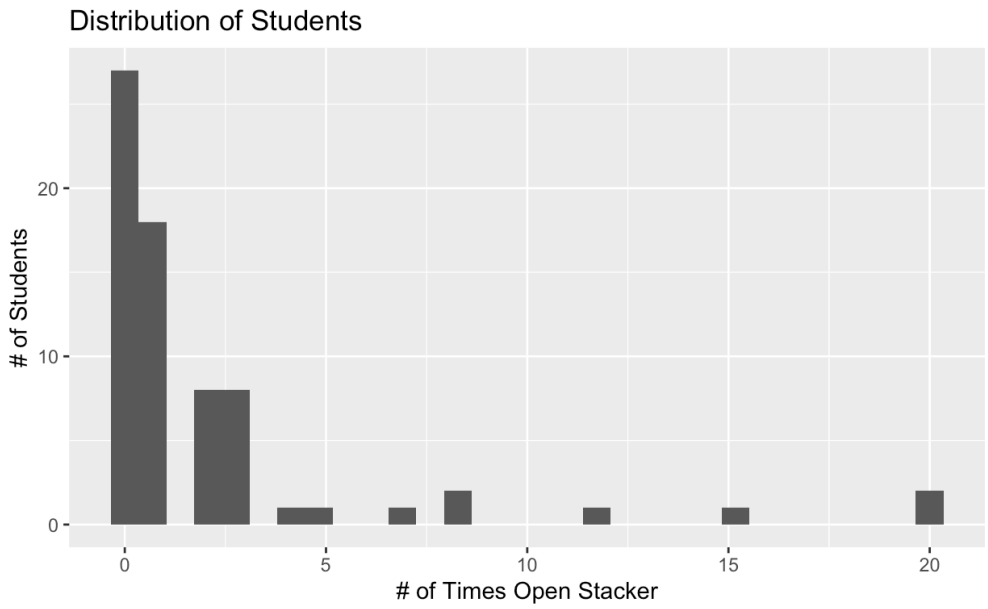


Figure 8.6: How often do students open Stacker?

students used the Stacker several times.

8.7.3 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 questions.

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 8.5, 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.

8.7.4 Enhancing Learning Retention

People forget knowledge over time, but repetition helps slow this process [18]. Delayed repetition is more effective than immediate repetition [7], and active recall—where learners apply knowledge—yields better retention than passive exposure [43].

The SMoL Tutor incorporates two strategies to support retention: (1) review tutorials (Section 8.1.2) and (2) encouraging students to download completed tutorials for easier review (Section 8.1.5).

Future work includes optimizing the Tutor to deliver review exercises at the ideal frequency and spacing for maximum retention.

8.7.5 Dependencies on Prior Knowledge

The SMoL Tutor assumes that students have an *intuitive* understanding of programming. While it may seem to require more extensive prior knowledge—especially since it uses technical terms like "environments" and "heap"—the Tutor carefully introduces such terms only when necessary to provide rigorous explanations.

Over the course of the tutorials, the SMoL Tutor introduces and explains the following technical terms: *definitions*, *variable definitions*, *bind*, *function definitions*, *formal parameters*, *call*, *actual parameters*, *blocks*, *values*, *errors*, *scope*, *shadowing*, *declarations*, *mutable values*, *vectors*, *(vector) elements*, *alias*, *heap*, *heap addresses*, *variable assignments*, *mutable variables*, *environments*, *closures*, and *lambda expressions*.

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. For example, the Tutor could be extended to introduce Greek-letter notation or to provide more concise explanations for fundamental concepts such as *variable definitions*.

8.7.6 Question Ordering

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

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

- Instructors often prefer to order questions 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 questions 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 questions by misconception, so students can focus on one misconception at a time. However, many questions 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 [7].

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 gather data for improving 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 can not accidentally harm students simply because I hold strong opinions about certain learning theories.

CHAPTER 9

RELATED WORK

This chapter is a work in progress, and some related work citations are yet to be included.

9.1 Tutoring Systems

There is an extensive body of literature on tutoring systems ([57] 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 [57] where the outer loop iterates through "tasks" (i.e., educational activities) and the inner loop iterates through UI events within a task. I follow the same structure in my tutor (Chapter 8).

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 (in computer science) that has this specific design.

9.2 Pedagogic Techniques

The SMoL Tutor 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 [37] 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 [45] have shown their effectiveness in multiple other domains.

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

9.3 Mystery Languages

My idea of misinterpreters is related to mystery languages [14, 36]. Both approaches use evaluators that represent alternative semantics to the same syntax. However, the two are complementary. In mystery languages, instructors design the space of semantics with pedagogic intent, and students must create programs to explore that space. Misinterpreters, in contrast, are driven by student input, while the programs are provided by instructors. The two approaches also have different goals: mystery languages focus on encouraging students to experiment with languages; misinterpreters aim at capturing students' misconceptions.

9.4 Misconceptions

Table 9.1: Similar misconceptions found in prior research.

Work	Population	Languages	Misconceptions
Fleury [21]	Unclear, likely CS2 students	Pascal	**CallerEnv**; **IsolatedFun**; **DefOrSet** (See their RULEs 2, 3a, and 3b in TABLE 2; RULE 1 doesn't apply to me.)
Goldman et al. [22, 23]	CS1 students	Java, Python, Scheme	Scope and memory model (See their Figures 4 and 5)
Fisler, Krishnamurthi, and Tunnell Wilson [20]	Third- and fourth-year undergrads	Java and Scheme	**FlatEnv**; **CallByRef**; **CallsCopyStruct**; **DefByRef** (See their Section 4)
Saarinen et al. [44]	CS2 students	Java	**StructByRef** (Their G2); **DefByRef** or **DefCopyStructs** (Their G3); **CallsCopyStruct** (Their G4).
Strömbäck et al. [51]	CS masters	Python	**FlatEnv**; **CallByRef**; **DefsCopyStruct** (See their section 4.2)
Strömbäck et al. [51]	CS undergrads	C++	**FlatEnv**; **CallByRef**; **DefsCopyStruct** (See their section 4.2)

Misconceptions related to scope, mutation, and higher-order functions have been widely identified in varying populations (from CS1 students to graduate students to users of online forums) over many years (since at last 1991) in varying programming languages (from Java to Racket) and in different countries (such as the USA and Sweden). Table 9.1 lists works that seem most relevant to me. In addition, Tew and Guzdial [53] also find several cross-language difficulties, though they do not classify them as misconceptions.

There are some small differences. Fleury [21] identifies a dynamic scope misconception that is different from **FlatEnv**:

CallerEnv Function values don't remember their environments. When a function is called,

the function body is evaluated in an environment that extends from the *caller's* environment.

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

Appendix A of [49] provides an extensive survey of misconceptions reported in research up to 2012. There are overlaps between my survey and theirs. For instance, my **CallerEnv** is their No. 47. However, because their descriptions are brief, it is difficult to tell whether a misconception in their survey matches my misconceptions. Because they provide neither misinterpreters nor representative program-output pairs, it is difficult to determine the overlaps precisely (showing the value of providing these two machine-runnable descriptions). At any rate, I certainly find no equivalent of **FlatEnv**, **DeepClosure**, and **DefByRef** in their survey.

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 How Good Are Misinterpreters at Modeling Misconceptions?

(Mis)interpreters are easier to write when the semantics are inductively defined on syntactic structure. As a result, defining misconceptions as misinterpreters inherently biases

them toward misconceptions that are consistent across contexts and independent of syntactic details. However, students' misconceptions do not always follow these patterns.

For example, misconceptions can vary across contexts. If students have learned how to swap the values of two variables, they are less likely to exhibit the DefByRef misconception when tested with a swap-like program than when tested with a very different program.

Misconceptions can also depend on syntactic details. In an earlier version of the Tutor, a function named `get-x` caused confusion because some students misread it as two separate variables, `get` and `x`. This misinterpretation was syntactically valid in SMoL's Lispy syntax—both `(get-x)` and `(get x)` were valid function calls—but led to very different interpretations.

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

10.3 Modifying SMoL Tutor to Teach a Different Semantics

The Tutor is designed to be compatible with languages that are not exactly SMoL but are close enough. For example, JavaScript does not produce an error when dividing a number by zero, whereas SMoL does. Despite this difference, the SMoL Tutor presents programs in JavaScript and relies on the division-by-zero error to diagnose misconceptions about evaluation strategy (e.g., eagerness). To address potential confusion, the Tutor displays a nearby warning: "JavaScript behaves differently."

In principle, it is possible to adapt the SMoL Tutor to teach a more substantially different semantics (e.g., R). However, since the SMoL Tutor was not originally designed with this flexibility in mind, such changes would likely require substantial modifications to the source code. The process involves several key steps:

1. Replacing the current reference interpreter with one that aligns with the new semantics.
2. Re-designating the old reference interpreter as a misinterpreter.
3. Re-running all programs to ensure that any incorrect answers are still mapped to at most one misinterpreter.

Additionally, the explanations provided by the SMoL Tutor must be updated to reflect the vocabulary and concepts of the new semantics.

10.4 Misconceptions and Programming Language Design

Studies of misconceptions can offer valuable insights into programming language design. However, language design is a complex discipline that should not be driven solely by the popularity of certain conceptions. See, for example, Tunnell Wilson, Pombrio, and Krishnamurthi [56] for in-depth discussion about designing programming languages based on programmer preference.

For example, dynamic scope (including FlatEnv) is widely regarded as problematic due to difficulties in managing variable access, which can lead to security vulnerabilities and other issues. Therefore, a programming language should avoid behaving like FlatEnv, regardless of how popular this conception is among its users.

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 lead to 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.

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

In Section 8.4.1 and Section 8.4.2, we observed that some questions are more difficult than others, even when they cover the same set of topics and are designed to detect the same misconception.

It is not surprising that question difficulty varies within the same tutorial. Some questions can reveal more misconceptions, offering students more potential ways to make a mistake.

What is more interesting, however, is that the same misconception is more likely to be detected by certain questions than others. This suggests that unknown or latent misconceptions may overlap with the targeted ones, offering more potential ways to make the mistake.

CHAPTER 11

CONCLUSION

This dissertation introduces the concept of misinterpreters, demonstrating their utility in identifying and defining misconceptions.

It also catalogs a collection of misconceptions related to SMoL characteristics.

Additionally, this work explores the design, implementation, and evaluation of SMoL Tutor and Stacker—two tools designed to enhance the teaching and learning of programming language behavior. Preliminary evidence suggests that SMoL Tutor is effective in addressing misconceptions.

APPENDIX A

INTERPRETERS

This appendix presents source code related to the interpreters. There is one definitional interpreter for the SMoL Language from Chapter 3, and one misinterpreter for each misconception from Chapter 6. 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

IsolatedFun Appendix A.13

Lazy Appendix A.14

NoCircularity Appendix A.15

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 IsolatedFun Misinterpreter

```
--- systems/smol-interpreters/Definitional.rkt  2025-02-08
   11:00:46
+++ systems/smol-interpreters/misinterpreters/IsolatedFun.rkt
   2025-02-11 11:41:13
@@ -215,7 +215,7 @@
       (let ([v ((eval-exp env) e)])
         ((env-update! env) x v))]
      [(Deffun f xs b)
-      (let ([v (function xs b env)])
+      (let ([v (function xs b the-primordial-env)])
         ((env-update! env) f v))]))
```

```
 (define (eval-body env xvs b)
@@ -239,7 +239,7 @@
     (type-case Expression e
       [(Con c) (embedded c)]
       [(Var x) (env-lookup env x)]
-      [(Lambda xs b) (function xs b env)]
+      [(Lambda xs b) (function xs b the-primordial-env)]
       [(Let xes b)
        (local [(define (ev-bind xv)
                  (let ([v ((eval-exp env) (snd xv))])
```

A.14 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.15 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

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

Introduce "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`

Evaluate 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.

Introduce "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

Introduce 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))
```

Functions are lambdas

```
(deffun (f x y z) body)
```

is a "shorthand" for

```
(defvar f (lambda (x y z) body))
```

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 et al. "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] Kaian Cai et al. “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).

[7] Nicholas J. Cepeda et al. “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).

[8] Kartik Chandra et al. *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).

[9] Luca Chiodini et al. “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).

[10] John Clements, Matthew Flatt, and Matthias Felleisen. “Modeling an Algebraic Stepper”. In: *Programming Languages and Systems*. Ed. by Gerhard Goos et al. 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).

[11] 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).

[12] *CodeMirror*. URL: http://codemirror.net/ (visited on 02/18/2025).

[13] *Color Wheel, a Color Palette Generator | Adobe Color*. URL: https://color.adobe.com/create/color-wheel (visited on 02/18/2025).

[14] Amer Diwan et al. "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).

[15] 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).

[16] 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).

[17] 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).

[18] 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).

[19] Matthias Felleisen et al. *How to Design Programs: An Introduction to Programming and Computing*. Mit Press, 2018. URL: https://books.google.com/books?hl=en&lr=&id=PahcDwAAQBAJ&oi=fnd&pg=PR5&dq=how+to+design+program&ots=asWb_o-Tcd&sig=lwleNjAxvPagBsm_HKpr32FKGJo (visited on 04/17/2025).

[20] 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 Educa-*

tion. 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.

[21] 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.

[22] Ken Goldman et al. "Identifying Important and Difficult Concepts in Introductory Computing Courses Using a Delphi Process". In: *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '08. New York, NY, USA: Association for Computing Machinery, Mar. 2008, pp. 256–260. ISBN: 978-1-59593-799-5. DOI: 10.1145/1352135.1352226.

[23] Ken Goldman et al. "Setting the Scope of Concept Inventories for Introductory Computing Subjects". In: *ACM Transactions on Computing Education* 10.2 (June 2010), 5:1–5:29. DOI: 10.1145/1789934.1789935. (Visited on 04/12/2023).

[24] 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).

[25] 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).

[26] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016. URL: https://books.google.com/books?hl=en&lr=&id=J2KcCwAAQBAJ&oi=fnd&pg=PR15&dq=practical+foundation+of+programming+language&ots=EZHbsQGlh9&sig=ywbEodOBSrJBERL455YskNPbfAA (visited on 01/31/2025).

[27] 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).

[28] Lisa C. Kaczmarczyk et al. “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).

[29] 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).

[30] *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).

[31] Kuang-Chen Lu et al. “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).

[32] Andrés Moreno et al. “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).

[33] 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).

[34] Joe Gibbs Politz et al. “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).

[35] Joe Gibbs Politz et al. “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).

[36] 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).

[37] George J. Posner et al. “Toward a Theory of Conceptual Change”. In: *Science education* 66.2 (1982), pp. 211–227. (Visited on 10/10/2023).

[38] 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).

[39] Helen Purchase. “Which Aesthetic Has the Greatest Effect on Human Understanding?” In: *Graph Drawing*. Ed. by Gerhard Goos et al. 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).

[40] Yizhou Qian and James Lehman. “Students’ Misconceptions and Other Difficulties in Introductory Programming: A Literature Review”. In: *ACM Transactions on Computing Education* 18.1 (Mar. 2018), pp. 1–24. ISSN: 1946-6226. DOI: 10.1145/3077618. (Visited on 01/31/2025).

[41] Teemu Rajala et al. “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).

[42] 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).

[43] 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).

[44] Sam Saarinen et al. “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.

[45] 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).

[46] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. 3rd. URL: https://www.plai.org/ (visited on 02/14/2025).

[47] 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).

[48] 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).

[49] Juha Sorva. “Visual Program Simulation in Introductory Programming Education”. PhD thesis. Aalto University, 2012. URL: https://aaltodoc.aalto.fi/handle/123456789/3534.

[50] 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).

[51] Filip Strömbäck et al. “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.

[52] C. Taylor et al. “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).

[53] 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).

[54] Preston Tunnell Wilson, Kathi Fisler, and Shriram Krishnamurthi. “Evaluating the Tracing of Recursion in the Substitution Notional Machine”. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. SIGCSE ’18. Baltimore

Maryland USA: ACM, Feb. 2018, pp. 1023–1028. ISBN: 978-1-4503-5103-4. DOI: 10.1145/3159450.3159479. (Visited on 01/30/2025).

[55] 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).

[56] 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).

[57] 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.

[58] *Wat*. URL: https://www.destroyallsoftware.com/talks/wat (visited on 01/23/2025).

[59] *WebAIM: Contrast Checker*. URL: https://webaim.org/resources/contrastchecker/ (visited on 02/18/2025).