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) a translator that enables the presentation of equivalent programs across multiple programming languages, (2) the idea that formally defining misconceptions allows for more effective identification and correction, and (3) 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

ABOUT THIS DRAFT

This document is a draft of my dissertation. The following issues are known, and I plan to address them before my defense:

- I will work with Shriram to correct/refine grammar throughout the document.
- I will extend the evaluation of SMoL Tutor (Section 8.2) as follows: by: (1) examining how well the dataset from the primary population (the PL course) correlates with other populations. Preliminary results from an older version of SMoL Tutor [20] suggest a correlation, but I have not yet analyzed datasets collected after recent UI updates; and (2) presenting qualitative data to further confirm the nature of misconceptions.
- SMoL Tutor's learning objectives, which are *rules of program behaviors* (RPB), should be evaluated using the criteria outlined in [9], which include consistency, accuracy, coverage, transferability, and several others.
- The appendix will include SMoL Tutor's content, including its RPBs.
- Section 6.2 will incorporate more related work, specifically: (1) research on students tracing programs and (2) studies on how people fixate on superficial features.
- Equivalence Tasks, a new type of SMoL Tutor task, have not been presented.

- The Related Work chapter (Chapter 9) is incomplete. I plan to add comparisons between Stacker (Chapter 7) and established tracing tools, including Python Tutor, Stepper, and UUhistle. Stacker is also heavily influenced by Clements and Krishnamurthi [3], which will be discussed.
- Diff Stacker, an extension of Stacker, is not yet included.
- The conclusion chapter needs more details.

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 variables are *not* aliased (at least by default).
- Mutable values (e.g., arrays) are 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 [19]) and deviation (such as the documented semantic oddities of JavaScript and Python [38, 14, 22, 23]). 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 10 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) [16]. 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 [16]. 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 Tracing

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

Tracing can be done in different ways:

- Automatically, using tools like debuggers or Python Tutor [15].
- Manually, using pen and paper [36].
- Manually within a computer environment, using interactive tools [3].
- A mix of the above, where students manually interact with an automated system [32].

This dissertation presents a tracing tool and associated learning activities.

2.3 Misconceptions and Mistakes

A **misconception** is an incorrect mental model. A **mistake**, on the other hand, is an incorrect action, which can happen for various reasons (e.g., a typo or a misclick).

Mistakes do not necessarily indicate misconceptions. 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 semantic misconceptions by identifying, defining, and correcting them.

2.4 Notional Machines

The term "Notional Machine" was probably first introduced by Du Boulay, O'Shea, and Monk [8], who defined it as "the idealized model of the computer implied by the constructs of the programming language." Later, Du Boulay [7] 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 [7] 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 [27, 31]) typically restrict the term to semantics. However, as Duran, Sorva, and Seppälä [9] 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."

2.5 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 ac-

cessible to students or even instructors without a formal PL background.

Duran, Sorva, and Seppälä [9] 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 [19], and Python employs an unusual scoping rule [23].

- 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

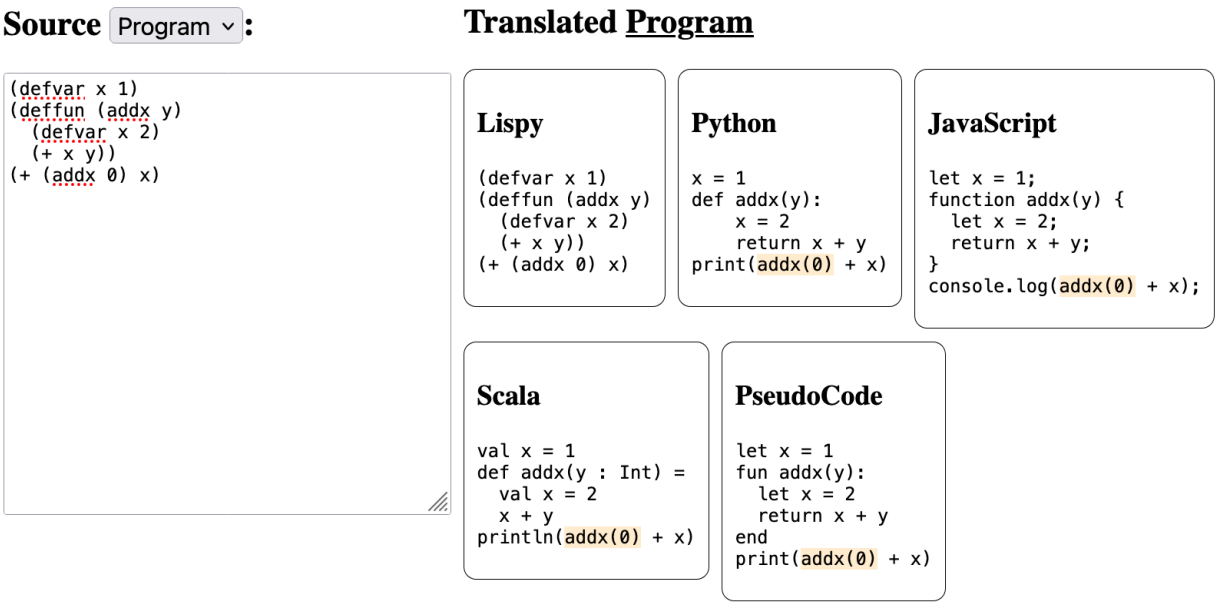


Figure 3.2: SMoL Translator Web App. Showing a translated program with the cursor hovering on one of the `addx(0)`.

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.

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.

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.

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).

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.

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.

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.

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).

Online Population A separate instance of the Tutor was published on the website of a programming languages textbook [30]. 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

A **misinterpreter** is an interpreter that executes programs incorrectly. Misinterpreters are closely related to misconceptions, which are often identified by observing how people interpret programs. If someone consistently misinterprets certain kinds of programs or provides an explanation that clearly differs from the correct semantics, we can infer a misconception.

Thus, misinterpreters provide a precise way to define misconceptions: a misinterpreter is, in essence, a definitional interpreter for a misconception.

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`.

Misinterpreters help address such ambiguity in identifying misconceptions in at least two ways:

- When implementing a misinterpreter for a misconception, we must make conscious design decisions, which can reveal ambiguities. In the example above, we must decide whether earlier definitions is shadowed or mutated.
- If multiple misconceptions are known, we can run a program through their misinterpreters to verify whether a wrong answer corresponds to exactly one misconception. If a wrong answer aligns with multiple misconceptions, we refine our test programs (as with the second program above). If no known misconception explains the answer, this suggests a previously unidentified misconception. For example, if we already had misinterpreters for the shadowing and mutation misconceptions, we would quickly realize that the first program is not ideal for distinguishing between them.

CHAPTER 6

SMOL MISCONCEPTIONS

Section 6.1 lists the misconceptions I identified.

Section 6.2 discusses related work on identifying misconceptions, and summarize approaches that are known to be good or bad.

Section 6.3 describes how I identified my collections of misconceptions.

6.1 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.

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

6.2 How (Not) to Identify Misconceptions?

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* [21]: 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* [17] (with a growing number for computer science, as a survey lists [34]). In terms of mechanics, a concept inventory is just an instrument consisting of multiple-choice questions

(MCQs), where each question has one correct answer and several wrong ones. However, the wrong ones are chosen with great care. Each one has been validated so that if a student picks it, we can quite unambiguously determine *what misconception the student has.* For instance, if the question is "What is `sqrt(4)`?", then `37` is probably an uninteresting wrong answer, but if people appear to confuse square-roots with squares, then `16` would be present as an answer.[1]

Aside, we'd better give identified misconceptions accurate descriptions using misinterpreters Chapter 5.

6.3 How I Identified the Misconceptions

The considerations discussed in Section 6.2 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.

6.3.1 Generating Problems Using Quizius

My main solution to the expert blind spot is to use the Quizius system [28]. 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.

[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.

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 [18] and uses that to choose a program.

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

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 [28] 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

[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.

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.

6.3.2 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.

6.3.3 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 6.2. In a concept inventory, each option must either be the correct answer or correspond to exactly one misconception. In my case, the one-to-one mapping is mostly, but not entirely, preserved:

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

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

It was important to add wrong answers that are not utterly implausible because those would become easy to eliminate. Therefore, I added extra options as follows:

1. The "error" option is added if it is not already an option.
2. Collect *templates* of existing options. A template of an option is the option with all the number literals removed. For instance, The template of "`2 3 45`" is "`_ _ _`".

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

I hope this reduced both guessing and elimination and forced students to actually think through the program. Of course, these new answers do not have a clear associated misconception.

CHAPTER 7

STACKER

Section 7.1 provides a guided tour of 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 Stacker.

Section 7.4 discusses design decisions.

7.1 A Guided Tour of Stacker

Stacker is a GUI application for tracing SMoL programs. Figure 7.1 shows the interface when Stacker is first opened. The UI consists of two main components, separated by a vertical gray bar:

(Program) Editor Panel For entering SMoL programs. This panel includes three bars at the top and a program editor, which supports common text-editing features like syntax highlighting. Section 7.2 provides more details.

Trace Panel For starting, stopping, and navigating traces. This panel includes an advanced

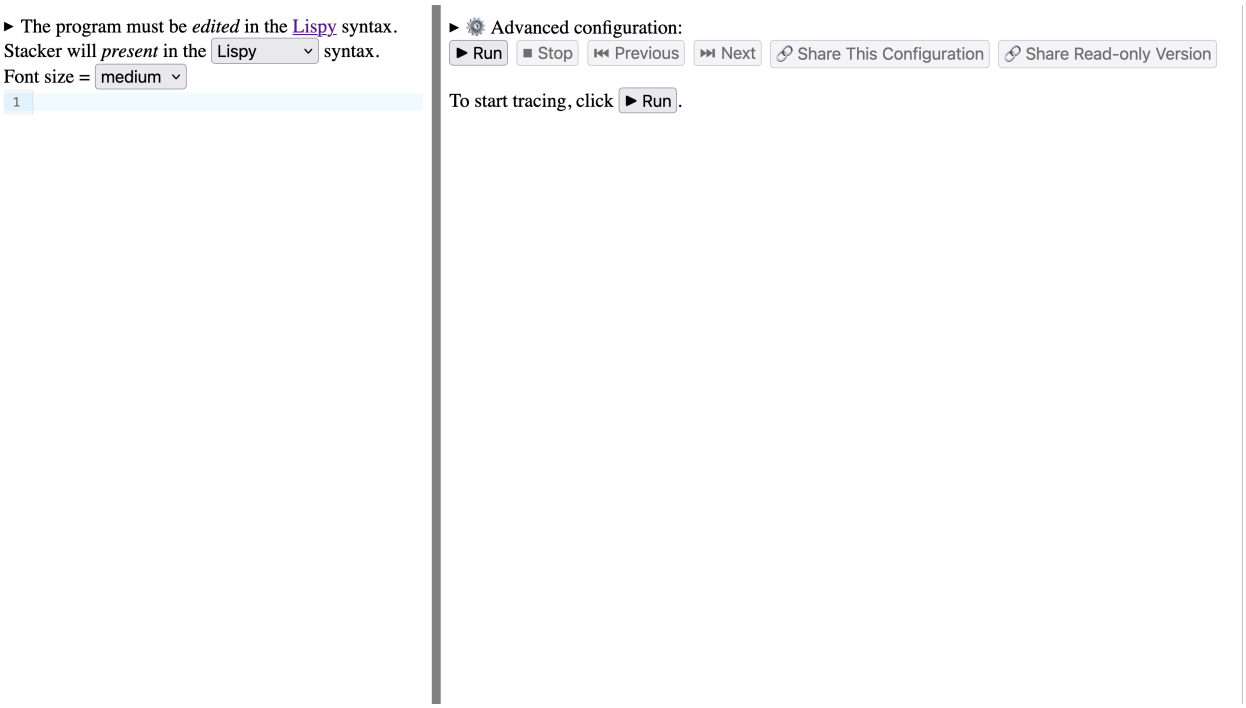


Figure 7.1: The Stacker user interface

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.

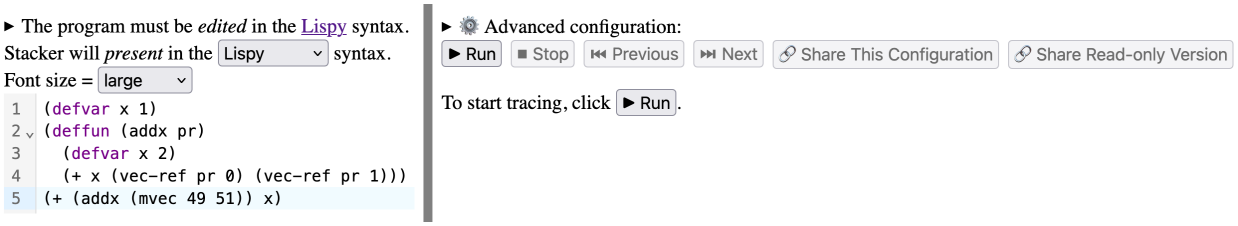


Figure 7.2: The Stacker user interface filled with an example program

7.1.1 Example Program

Figure 7.2 shows Stacker loaded with the following SMoL program:

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

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

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

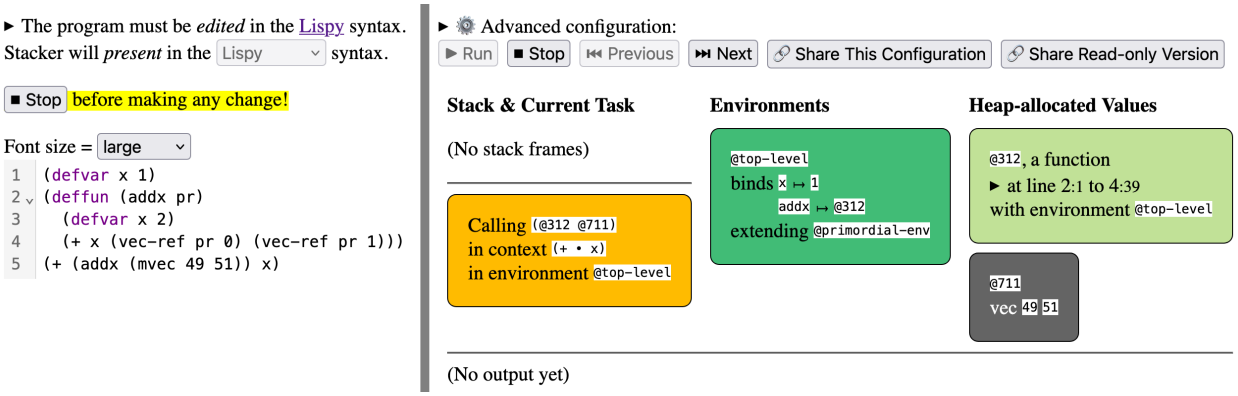


Figure 7.3: An example Stacker trace (showing state 1 out of 5)

7.1.2 Running the Program

Clicking one of the "run" buttons in Figure 7.2 transitions 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. 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.

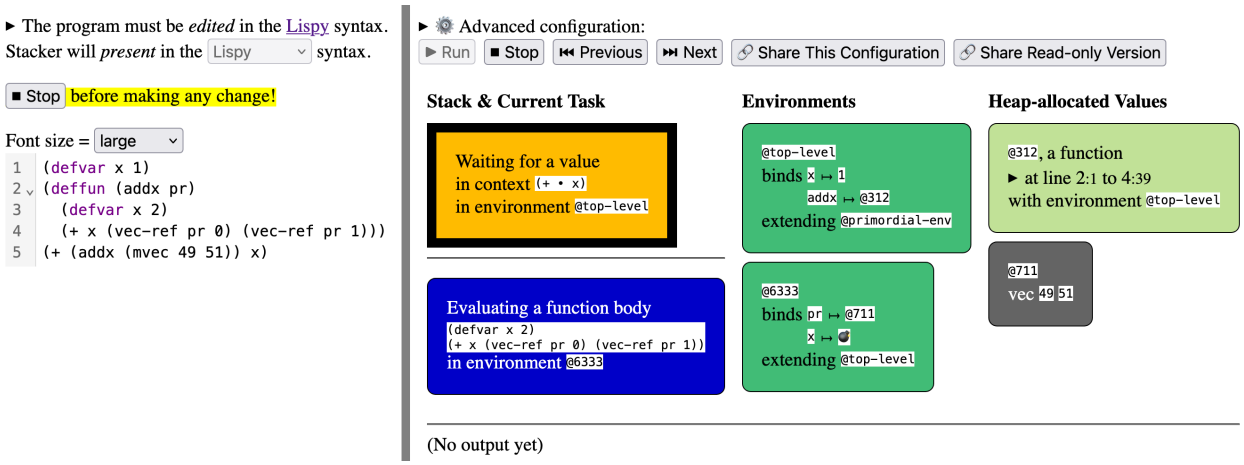


Figure 7.4: An example Stacker trace (showing state 2 out of 5)

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. 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".

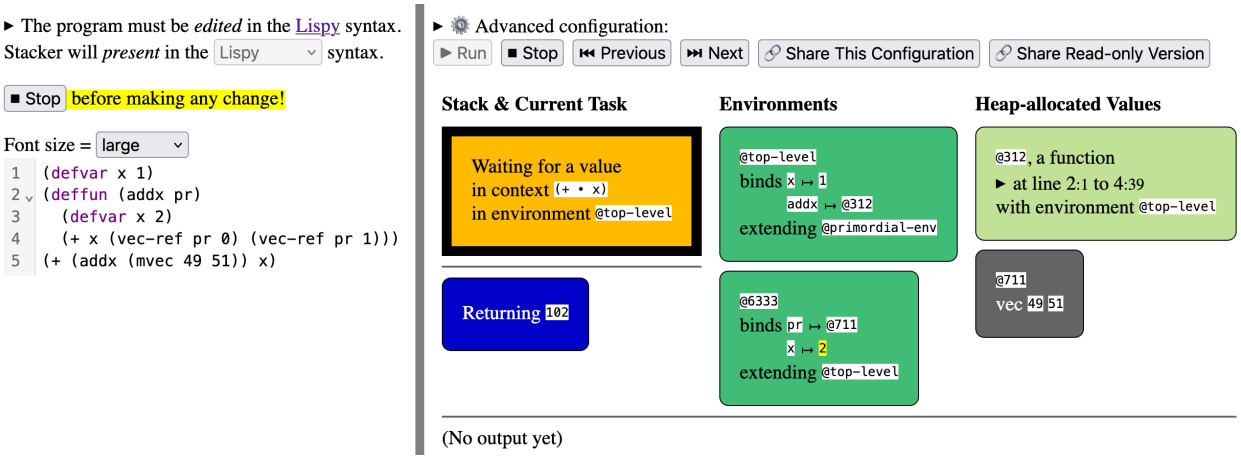


Figure 7.5: An example Stacker trace (showing state 3 out of 5)

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

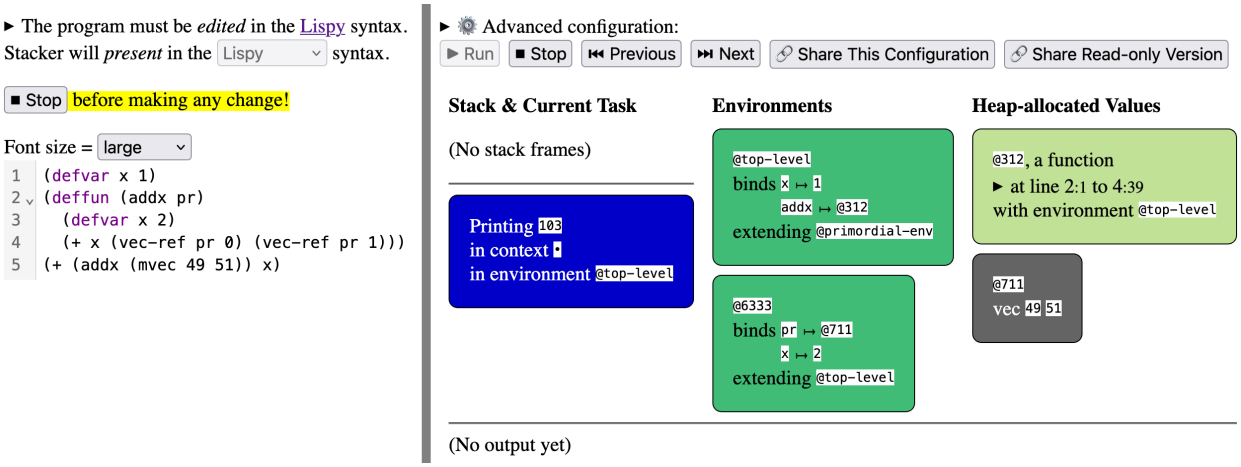


Figure 7.6: An example Stacker trace (showing state 4 out of 5)

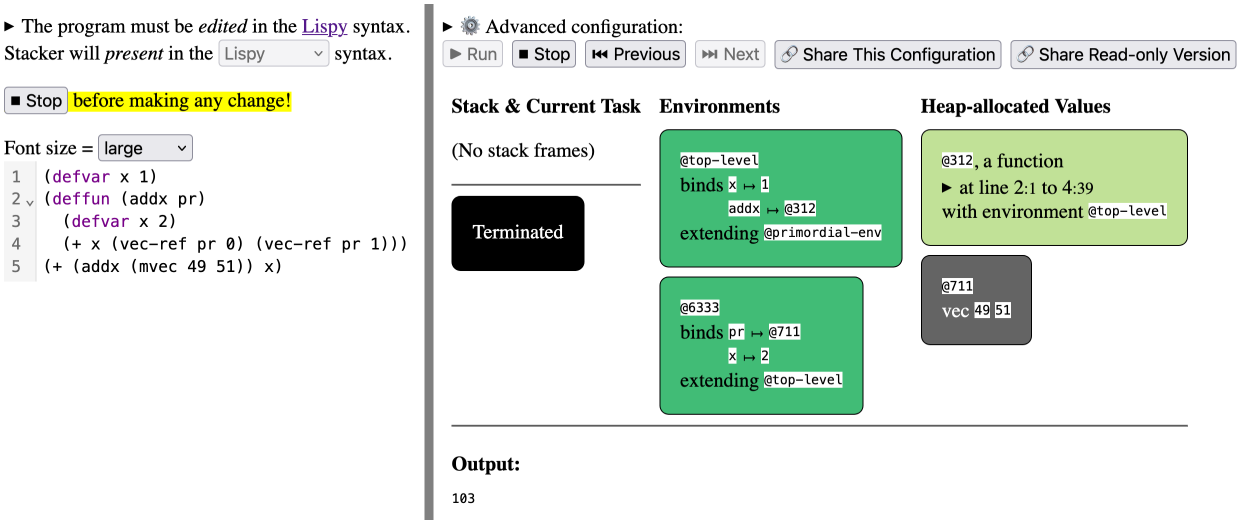


Figure 7.7: An example Stacker trace (showing state 5 out of 5)

5. The stacker is mutating a data structure.

6. The stacker is printing a value.

7. The stacker just terminated.

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 Stacker:

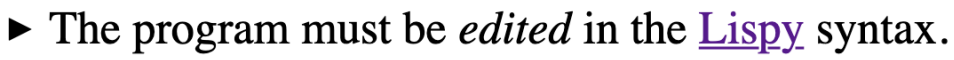
- Stacker can present the program and the trace in any language supported by the SMoL Translator (Chapter 3), but programs must be edited in the SMoL Language.
- Every (non-final) state presents enough information to predict the next state.
- Users can create a sharable URL for the current state, allowing others to view the exact same trace.
- 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

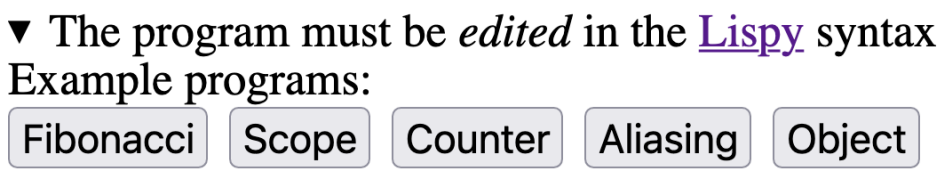
This section presents more details on using Stacker, complementing the guided tour (Section 7.1).

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).

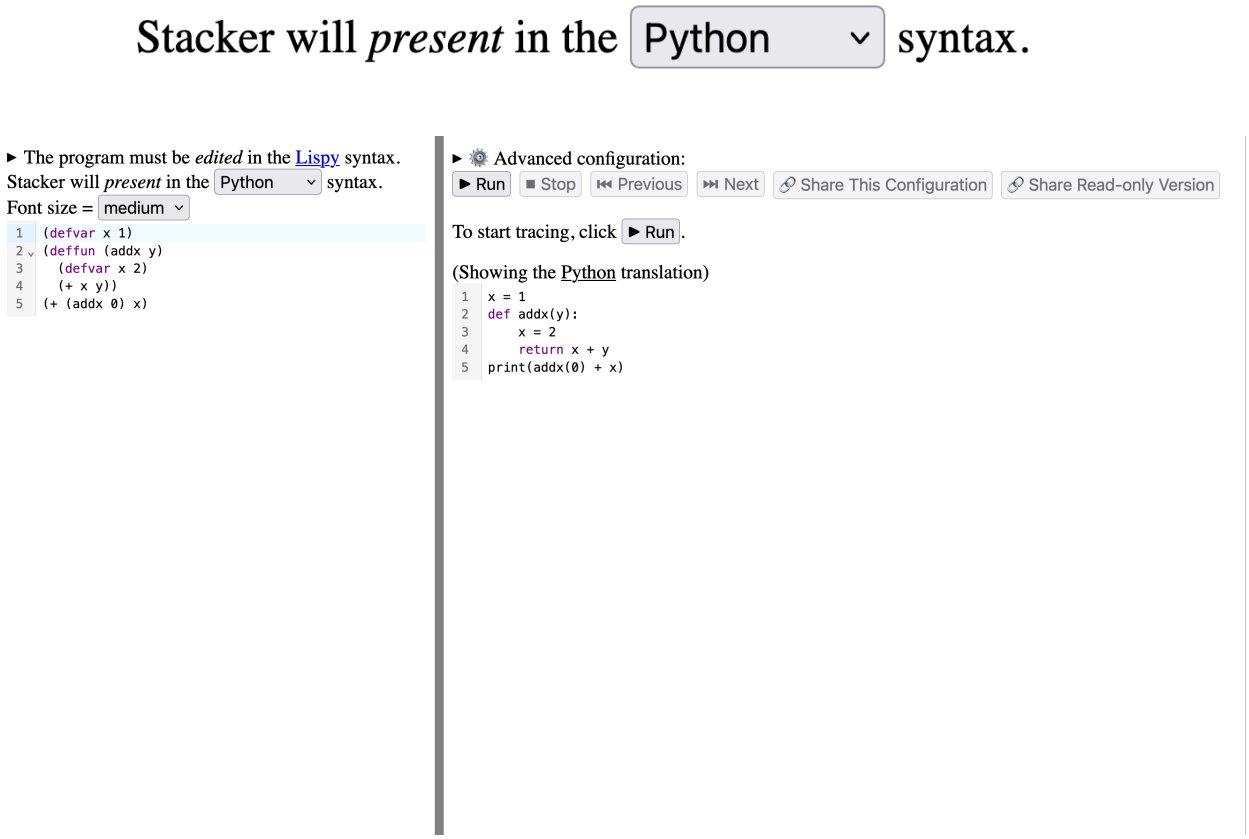
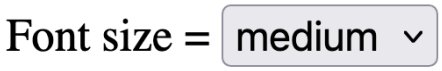


Figure 7.8: The live translation feature of Stacker. When users are editing their programs and the presentation syntax is set to non-Lispy, a live translation of the program is shown on the trace panel.

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:

- 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 [4], 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 Stacker in a simplified mode, hiding irrelevant details for easier trace navigation (Figure 7.9).

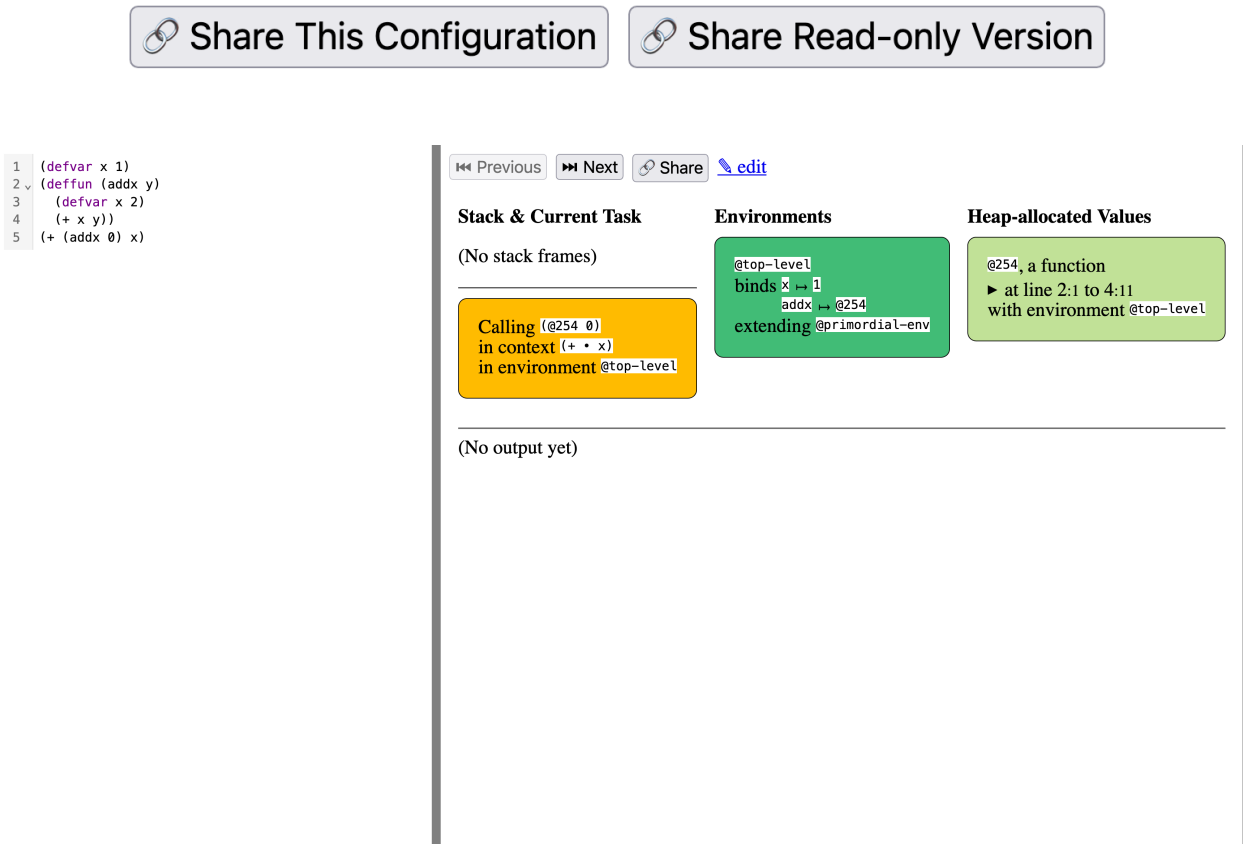


Figure 7.9: A read-only view of a Stacker trace. This kind of view is creatd by "Share Read-only Version" or the "Share" button in a read-only view.

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, 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

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

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 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 [26] 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.

Figure 7.10: A state-to-output question

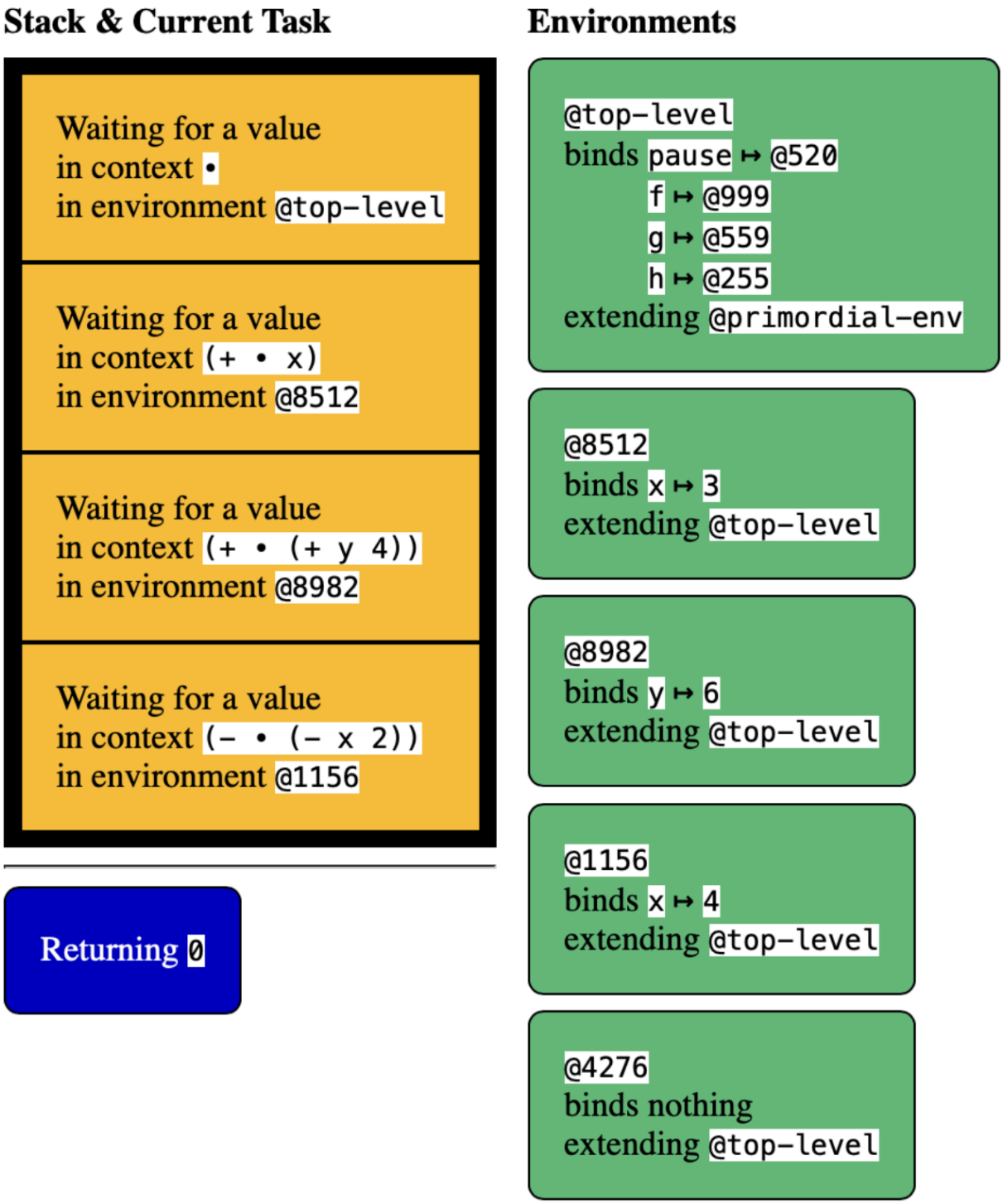
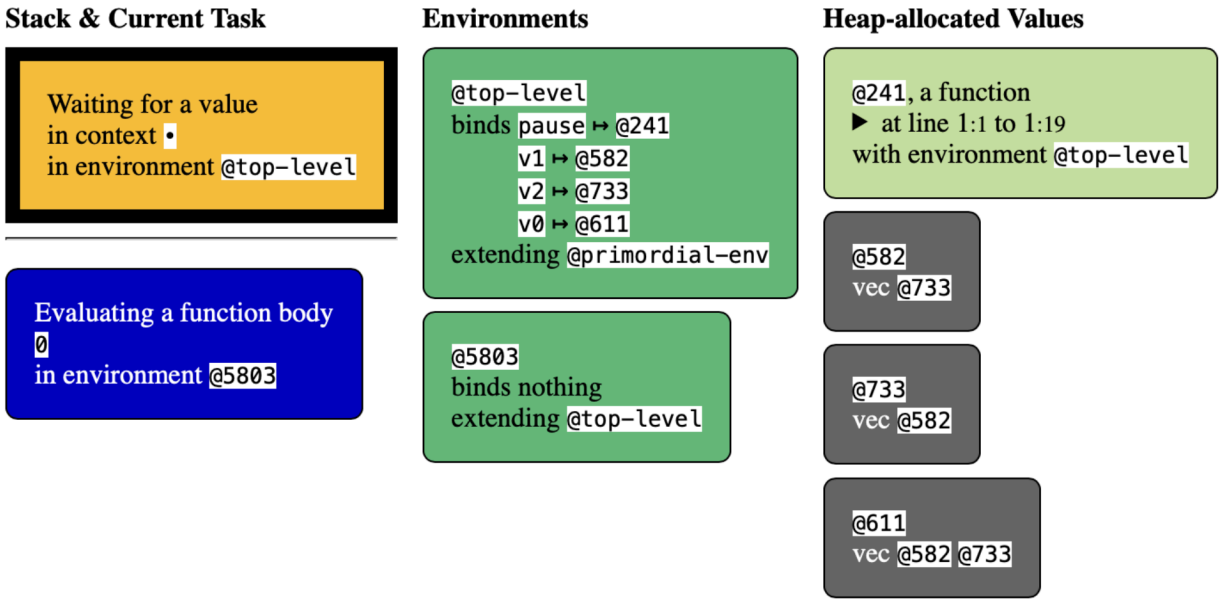


Figure 7.11: A state-to-program question



7.3.5 Using Stacker to Teach Generators

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

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

This section discusses key design decisions, considering alternative approaches and related work.

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()
```

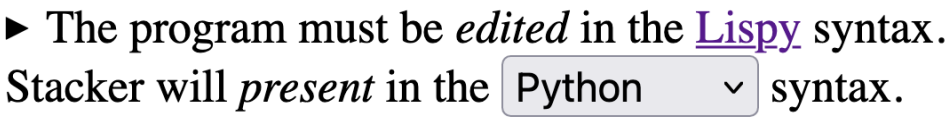
7.4.1 Amount of Information in States

Many tracing applications do not display enough information to predict future states. Stacker does (modulo randomness of heap addresses). This design allows instructors to assign tasks like those in Sections 7.3.3 and 7.3.4.

An additional benefit of displaying enough information is that if students fully understand Stacker, they fully understand the language's behavior. In this sense, Stacker itself serves as a learning objective.

7.4.2 Editing and Presenting Languages

While Stacker supports viewing traces in multiple languages, users must write programs in the SMoL Language. This distinction can be confusing. To mitigate this, Stacker emphasizes the words "edit" and "present" in the UI:



An alternative approach would allow program editing in other supported languages, but this presents two challenges. First, maintaining multiple language parsers requires significant engineering effort. Existing parsers cannot be directly reused because many language constructs fall outside the SMoL Language, and even supported constructs may not map cleanly (e.g., Python's `nonlocal` and `global` keywords). Second, allowing users to edit programs in different languages risks misinterpretation–users may mistakenly assume Stacker accurately traces those languages. This confusion already exists to some extent, which Stacker mitigates by using "syntax" rather than "language" in the UI. However, allowing direct editing in, for example, Python, would further increase this risk.

Another alternative is a multi-syntax structural editor, which provides a perhaps more user-friendly editing experience and avoids the engineering effort compared to full multi-language editing.

7.4.3 Sharing URLs

Currently, trace information is embedded in the URL as parameters. An alternative design would store the trace on a server and use a storage ID in the URL instead. While this approach results in shorter URLs, it adds server maintenance overhead and cannot guarantee permanence of the URLs due to storage constraints.

7.4.4 Presentation of Memory References

Stacker represents memory references with addresses, unlike Python Tutor [15], which uses arrows from source to target.

Arrows are intuitive for simple references. However, as complexity increases, overlapping arrows create visual clutter (compare Figures 7.13 and 7.14). Address-based representation avoids this problem, but tracking references remains cognitively demanding, and its complexity, like arrows, also scales with the number of entities.

To mitigate this, Stacker highlights referred entities (Section 7.2.8), but this may not be as visually clear as arrows. A hybrid design could be ideal:

- When a reference is hovered, arrows appear pointing to the referred entity.
- When a box is hovered, arrows appear pointing to its references.

Figure 7.13: Python Tutor presenting a cyclic data structure

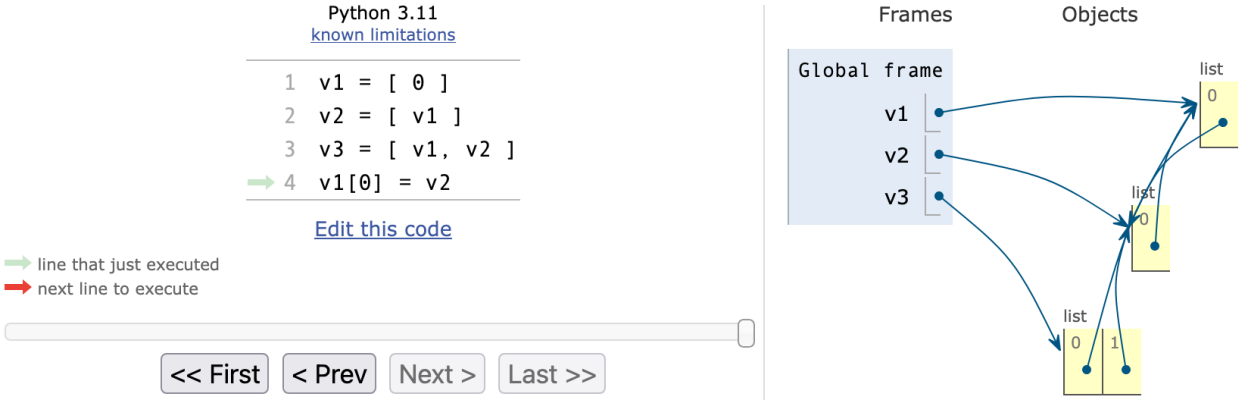
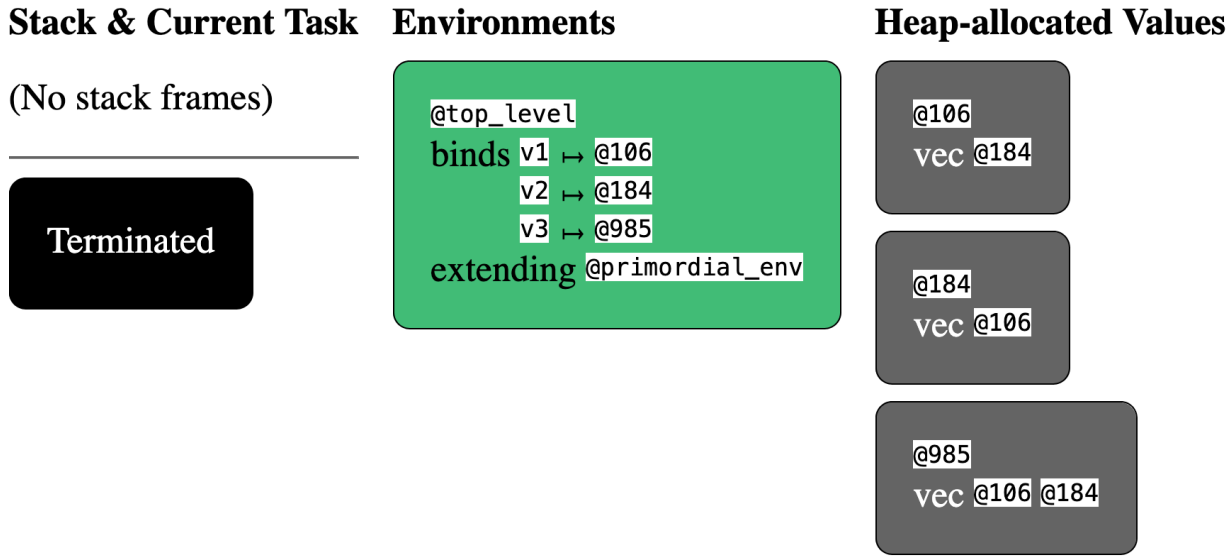


Figure 7.14: Stacker presenting a cyclic data structure



7.4.5 Accessibility

Stacker's web-based design makes font size adjustments easy.

The color palette (Table 7.1) is designed for accessibility:

- Background colors are verified with Adobe Color [5] for colorblind-friendliness.
- Most foreground-background combinations meet WCAG 2.0 level AAA contrast standards (verified via WebAIM [39]). The only exception is white on #646464, which meets level AA.

7.4.6 Web-based Application vs Traditional Application

A web-based approach offers several advantages:

No setup required Students only need a web browser to access Stacker.

Built-in accessibility Font sizes can be adjusted freely (Section 7.4.5).

Easy to implement sharing Trace information can be shared as URL parameters.

Editable traces Since traces are HTML documents, users can modify them using browser developer tools.

7.4.7 Prerequisite Knowledge

There is a trade-off between accurately representing program execution and minimizing prerequisite knowledge. Using standard terminology ensures authenticity but may confuse students unfamiliar with certain concepts, requiring additional instructor explanation.

How to balance the trade-off depends highly on the teaching context, so there is surely no perfect solution. Stacker currently avoids dependencies on system-level courses by making minor adjustments:

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

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 8.2 evaluates the tutor, demonstrating its effectiveness in correcting certain misconceptions. Section 8.3 explores the design space.

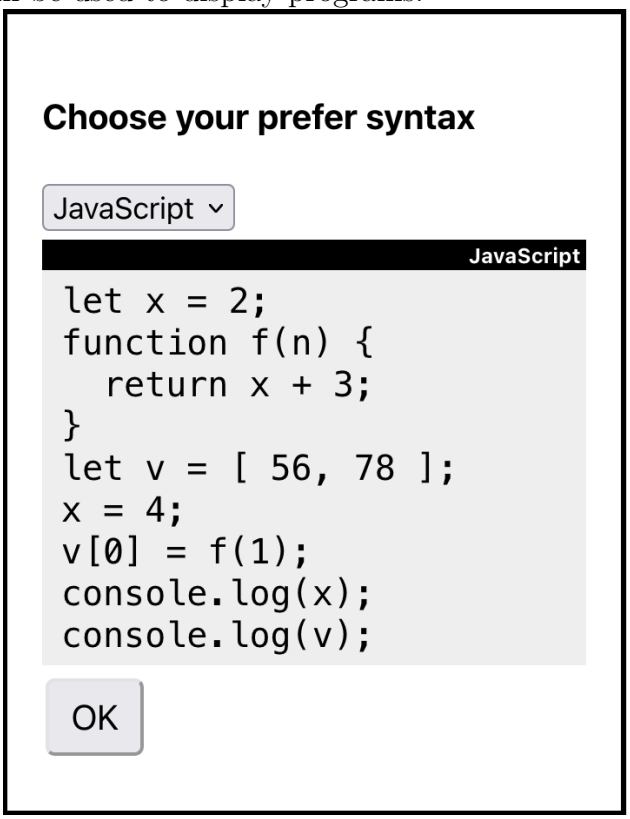
8.1 A Guided Tour of SMoL Tutor

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

Table 8.1: SMoL Tutor topics

Topic
def{1-3}: variable and function definitions
vectors{1-3}: mutable compound data
mutvars{1-2}: mutable variables
lambda{1-3}: first-class functions and lambda expressions

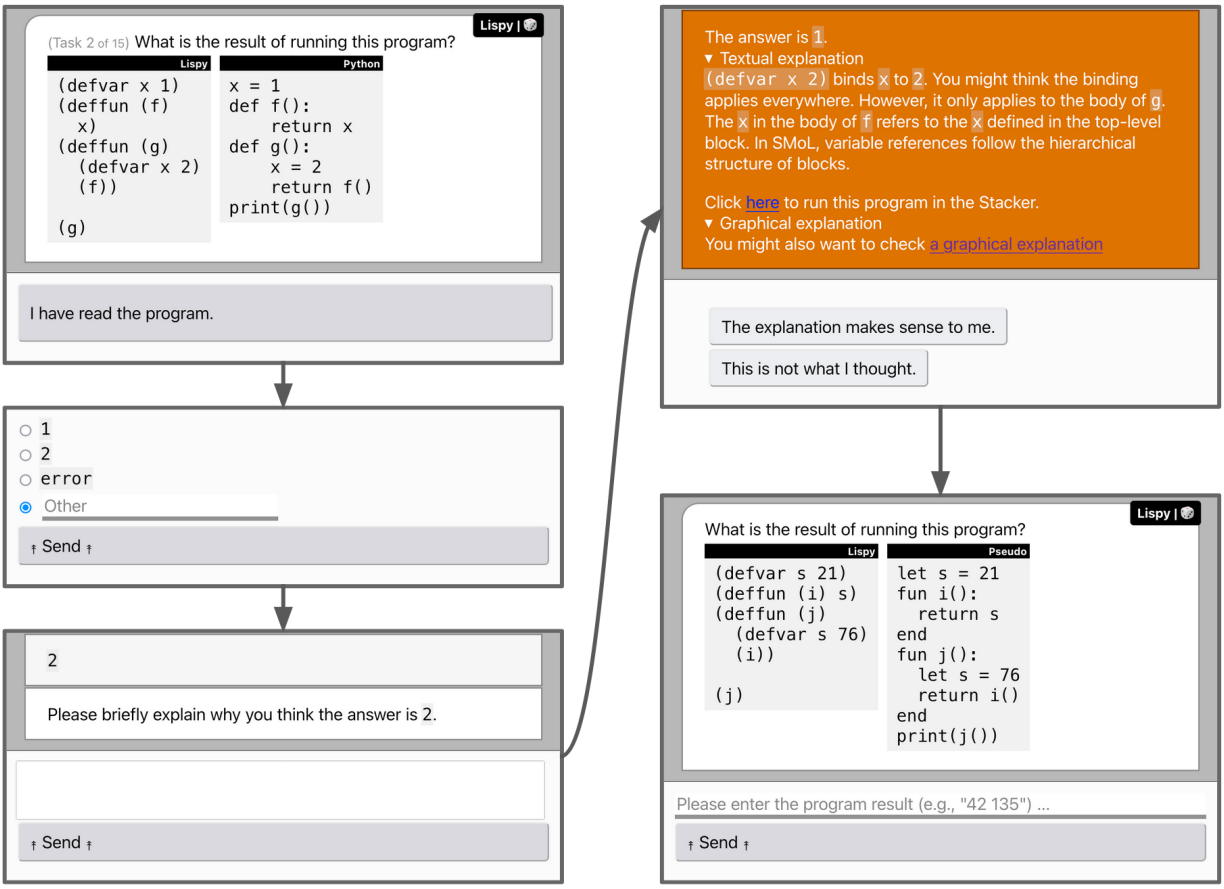
After selecting a syntax, users must also choose a tutorial. The Tutor covers five major topics, listed in Table 8.1. Each topic is further divided into 2-3 tutorials, named using labels such as **def1**. The intended duration for each tutorial was 20-30 minutes, but my data show that students typically 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.

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) and (b) are then presented with a slightly modified follow-up question.

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

Figure 8.2: An Interpreting Task in SMoL Tutor.



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 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.

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

- 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.

Additionally, many tutorials present learning objectives, formatted as **rules of program behavior**.

8.2 Evaluation: How Effective is SMoL Tutor?

I collected data from all the populations described in Chapter 4. This section focuses on analyzing the PL course population.

Effective training should reduce the likelihood of trainees repeating the same mistakes as they progress. In the context of SMoL Tutor, where training primarily involves interpreting tasks, repeating mistakes means selecting incorrect answers that reflect the same misconception. Thus, for each misconception, we examine whether students become less likely to choose answers associated with that misconception as they complete more related tasks.

There are a few additional considerations when translating this idea into a statistical analysis:

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.

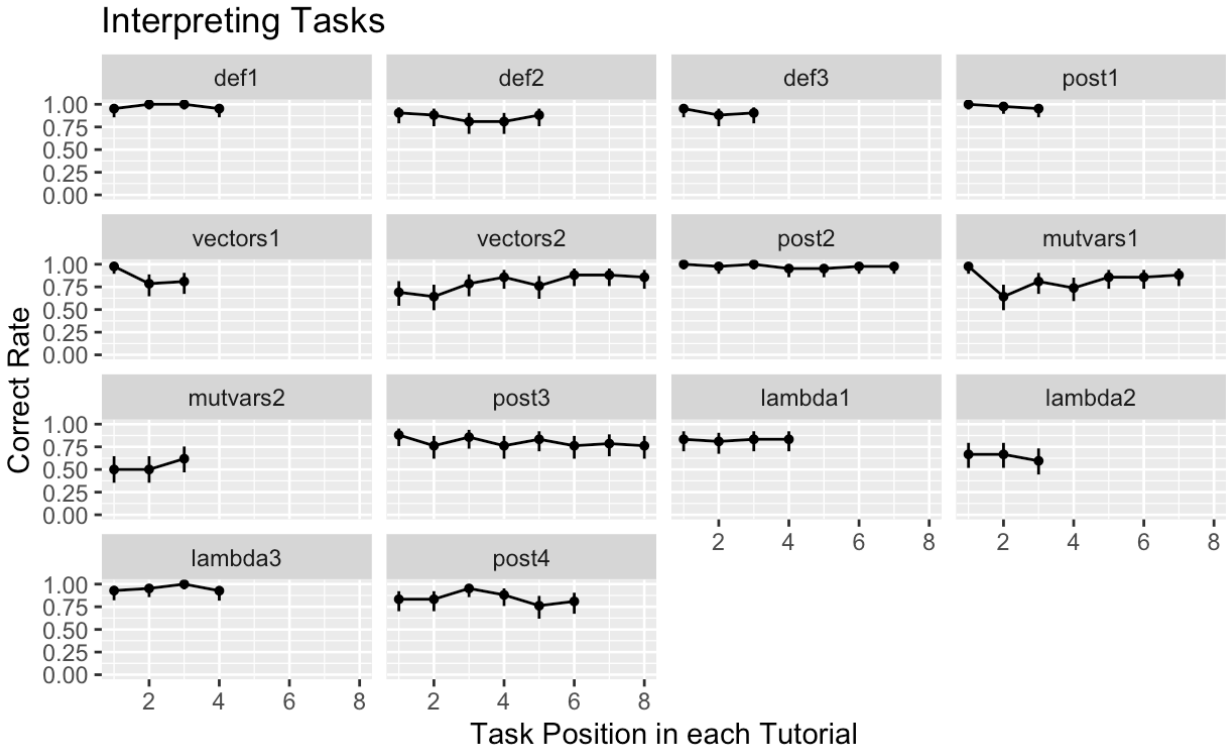
2. The difficulty of interpreting tasks varies, and the randomization of question order may not be perfectly balanced. If tasks happen to be ordered in a particular way (e.g., if difficult tasks are disproportionately placed at the beginning for most students), it could create a false impression of learning.

The first concern seems unlikely. Figure 8.3 shows students' performance across all interpreting tasks. If they were merely learning the UI, we would expect a clear upward trend across all tutorials. However, no such trend is apparent.

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

Task-Specific Some tasks are inherently more difficult than others, regardless of their order.

Figure 8.3: Students' performance in all interpreting tasks.



Position-Specific Tasks earlier in the sequence are systematically easier or harder than later ones.

Both Task and Position Matter Both task difficulty and position influence student performance.

It is possible that all theories hold to some extent or that none provide a strong explanation. What we are particularly interested in is whether (1) the **Position-Specific** or **Both Task and Position Matter** theories are at least as plausible as the **Task-Specific** theory and (2) in all position-related theories, the effect of position is clearly negative on students' probability of choosing answers associated with that misconception.

Table 8.2 summarizes the analysis described above. The first column lists the misconceptions. The three AIC columns compare the *Akaike Information Criterion (AIC)* of the three theoretical models, each represented as a generalized linear model with a logit link function (since the outcome is binary—whether a student selected the wrong answer corresponding

Table 8.2: Are students learning from SMoL Tutor?

Misconception	***AIC***			***Position***	
	Task	**Position**	**BothMatter**	**Effect**	***p*-value**
DefByRef	54.53	51.30	51.99	-2.1041	0.054
CallByRef	60.06	61.08	61.94	0.2498	0.725
StructByRef	108.33	110.43	103.81	-1.2714	**0.014***
DefCopyStruct	66.52	68.26	67.53	-0.6624	0.330
CallCopyStruct	93.20	88.13	89.91	-1.3083	**0.024***
StructCopyStruct	90.55	107.69	90.58	-0.7560	0.166
FlatEnv	89.35	114.62	89.27	0.6948	0.161
DeepClosure	177.81	198.77	179.78	0.0636	0.875
DefOrSet	107.05	119.49	108.83	-0.2255	0.635
FunNotVal	97.53	95.87	99.34	-0.1137	0.736
Lazy	79.16	78.05	80.84	0.3026	0.442
NoCircularity	59.93	54.53	54.83	-2.4446	**0.014***

to the misconception). 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.

The two position-effect columns show the influence of task position on student performance. Since we have two models incorporating position effects, these columns report values from the model with the lower AIC. We expect a negative and statistically significant effect, meaning that students improve as they progress through tasks.

Key observations from Table 8.2:

- The Task-Specific model is never clearly superior to all other models.
- The position effect is typically negative.
- When the position effect is statistically significant (p-value ≤ 0.05), it is always negative.

These results suggest a partial success.

8.3 The Design Space Around SMoL Tutor

This section explores the design space surrounding SMoL Tutor, including past design iterations, the rationale behind changes, and alternative designs that were considered but not implemented.

8.3.1 SMoL Tutor UI Updates

Over time, 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 8.2).

8.3.2 SMoL Quizzes, the Precursor of SMoL Tutor

Before developing 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). SMoL Tutor, in contrast, provides links for students to run programs in the Stacker and logs those events. Figure 8.4 shows how often students run programs in the Tutor. Some students used the Stacker several times.

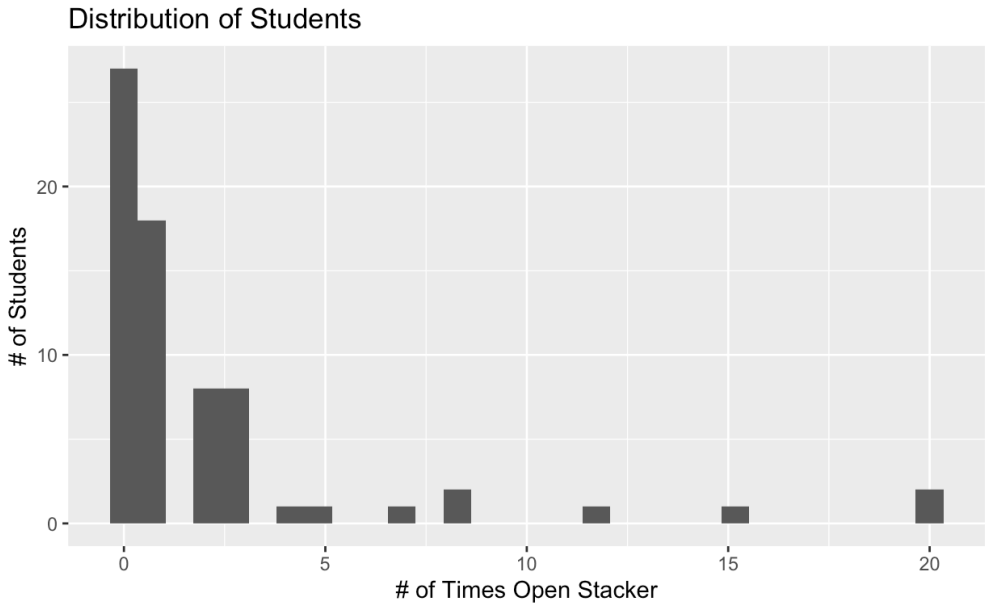


Figure 8.4: How often do students open Stacker?

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 ([37] 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 [37] 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 [2]), and in the process undoubtedly address some program behavior misconceptions. 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 [25] 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 [29] 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). [1] 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 [6, 24]. 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 [11]	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. [12, 13]	CS1 students	Java, Python, Scheme	Scope and memory model (See their Figures 4 and 5)
Fisler, Krishnamurthi, and Tunnell Wilson [10]	Third- and fourth-year undergrads	Java and Scheme	**FlatEnv**; **CallByRef**; **CallsCopyStruct**; **DefByRef** (See their Section 4)
Saarinen et al. [28]	CS2 students	Java	**StructByRef** (Their G2); **DefByRef** or **DefCopyStructs** (Their G3); **CallsCopyStruct** (Their G4).
Strömbäck et al. [33]	CS masters	Python	**FlatEnv**; **CallByRef**; **DefsCopyStruct** (See their section 4.2)
Strömbäck et al. [33]	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 [35] also find several cross-language difficulties, though they do not classify them as misconceptions.

There are some small differences. Fleury [11] 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 [32] 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

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
   2025-02-08 11:00:46
@@ -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).

BIBLIOGRAPHY

[1] 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).

[2] 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).

[3] 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).

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

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

[6] 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).

[7] 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).

[8] 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).

[9] 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).

[10] Kathi Fisler, Shriram Krishnamurthi, and Preston Tunnell Wilson. “Assessing and Teaching Scope, Mutation, and Aliasing in Upper-Level Undergraduates”. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 213–218. ISBN: 978-1-4503-4698-6. DOI: 10.1145/3017680.3017777.

[11] 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.

[12] 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.

[13] 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).

[14] 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).

[15] 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).

[16] 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).

[17] 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).

[18] 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).

[19] *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).

[20] Kuang-Chen Lu and Shriram Krishnamurthi. "Identifying and Correcting Programming Language Behavior Misconceptions". In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA1 (Apr. 2024), pp. 334–361. ISSN: 2475-1421. DOI: 10.1145/3649823. (Visited on 02/05/2025).

[21] 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).

[22] 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).

[23] 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).

[24] 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).

[25] 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).

[26] 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).

[27] 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).

[28] 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.

[29] 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).

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

[31] 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).

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

[33] 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.

[34] 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).

[35] 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).

[36] 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).

[37] 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.

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

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