LambdaLab:
An Interactive λ-Calculus Reducer for Learning

Daniel Sainati
Cornell University
USA
dhs253@cornell.edu

Adrian Sampson
Cornell University
USA
asampson@cs.cornell.edu

Abstract

In advanced programming languages curricula, the λ-calculus often serves as the foundation for teaching the formal concepts of language syntax and semantics. LambdaLab is an interactive tool that helps students practice λ-calculus reduction and build intuition for its behavior. To motivate the tool, we survey student answers to λ-calculus assignments in three previous classes and sort mistakes into six categories. LambdaLab addresses many of these problems by replicating the experience of working through β-reduction examples with an instructor. It uses visualizations to convey AST structure and reducible expressions, interactive reduction to support self-guided practice, configurable reduction strategies, and support for encodings via a simple macro system. To mimic informal, in-class treatment of macros, we develop a new semantics that describes when to expand and contract them. We use case studies to describe how LambdaLab can fit into student workflows and address real mistakes.

CCS Concepts • **Social and professional topics → Computing education**; • **Theory of computation → Lambda calculus**;

Keywords education, programming languages curriculum, educational tools, formal semantics, λ-calculus

ACM Reference Format:
Daniel Sainati and Adrian Sampson. 2018. LambdaLab: An Interactive λ-Calculus Reducer for Learning. In *Proceedings of the 2018 ACM SIGPLAN SPLASH-E Workshop (SPLASH-E '18), November 5, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3310089.3313180

1 Introduction and Motivation

The λ-calculus [3] is the first topic on the syllabus for many theoretical programming languages courses. Its simple structure provides a substrate for learning about formal grammars and operational semantics, and its universality illustrates the relationship between formalized core calculi and real-world programming languages. By learning the λ-calculus, students build intuition that will undergird all the remaining topics in a curriculum on formal semantics.

For the uninitiated, however, the λ-calculus is a tall hurdle. Students need to simultaneously master substitution-based evaluation, higher-order programming, and encodings, all while dealing with an unfamiliar syntax. Even students who understand the basic evaluation rules can fail to apply them to larger programs when they lack the underlying intuition. When new students work with the λ-calculus on paper, they face a minefield of potential formal mistakes that are difficult to disentangle from each other. An incorrect encoding for an arithmetic function, for example, may stem from a logic error, a substitution mistake, incorrect parenthesization, or incorrect application of the reduction rules—and the root cause can be difficult to diagnose.

This paper posits that part of the difficulty arises from the limited bandwidth for practice that manual λ-calculus reduction affords. While we can work through one example at a time in class, and students can theoretically write out evaluations alone for practice, the long latency means that students, in practice, see only a few completely worked examples. We advocate for integrating automated reduction into the theoretical PL curriculum to increase the rate at which students can try out examples, observe reduction results, and critique their own work. In our experience, this approach helps students avoid getting bogged down in low-level mechanical mistakes while helping them develop the intuition they need to understand more advanced topics that build on understanding of the λ-calculus. The opportunity for increased practice and improved intuition outweighs the risk of overreliance on automation: in the same way that graphing calculators can aid learning in math classes, reduction tools can help students be more ambitious in their learning of programming language formalisms.

This paper has two main components: an empirical investigation into the kinds of mistakes students make when

SPLASH-E '18, November 5, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6680-9/18/11…$15.00
https://doi.org/10.1145/3310089.3313180

learning the λ-calculus, and the design of an interactive tool that can help address these mistakes.

The investigation studies homework problems from two courses, an undergraduate elective and a PhD class, that teach the λ-calculus early in a curriculum about formal language semantics (Section 2). We find that a majority of errors stem from mechanical problems that represent misunderstandings of a variety of aspects of the λ-calculus. Access to automated reduction would help dispel most of these misunderstandings, as students would be provided with immediate feedback on the correctness of their reductions, preventing misunderstandings from compounding and reinforcing themselves due to repetition.

We develop a new interactive λ-calculus evaluator tool, called LambdaLab, that addresses these pitfalls (Section 3). Crucially, we find that a simple λ-calculus interpreter would not suffice to bring clarity to several of the misunderstandings, so we build LambdaLab with features that bring it closer to simulating the experience of working out examples on the whiteboard with the instructor. We use visualizations to help clarify the relationship between the linear token strings we write out and the ASTs we use for reduction. We emphasize the differences between different evaluation orders by visually indicating the "active redex" throughout a reduction sequence. Most significantly, we allow macros that make examples more intelligible, especially ones that focus on Church encodings. We develop a new operational semantics that describes when to expand macro definitions for each of four main reduction strategies, and we formalize a strategy for *contracting* macro definitions in reduction results that mimics the informal process that instructors follow in whiteboard examples (Section 4). Finally, we discuss how these features allow LambdaLab to improve student understanding via case studies that illustrate how how students can use it to address real homework problems (Section 5).

LambdaLab is open source and available online.[1] Our initial experience suggests that it is ready for integration into any curriculum that relies on the λ-calculus.

2 Measuring λ-Calculus Mistakes

We conducted an empirical investigation to study the kinds of difficulties that students encounter when learning the λ-calculus. Our goal is to understand how well automating reduction might address the mistakes that students actually make. We focus on three research questions:

1. What are the most common kinds of mistakes students make when learning the λ-calculus?
2. What proportion of these mistakes fall into categories that automating reduction can address?
3. How might a student use an automated reduction tool to identify and correct problems in their understanding of the λ-calculus?

To answer these questions, we analyze real, anonymized homework solutions from two previous PL courses. We describe our methodology for the empirical investigation, the categories of misunderstandings we found, and our conclusions regarding the utility of an interactive reduction tool.

2.1 Setup

We collected anonymized homework submissions from two programming languages courses at Cornell University: an advanced undergraduate elective (CS 4110) and a graduate course taken by most PhD students (CS 6110). Both courses focus on formal semantics and introduce the λ-calculus early in the semester.

We selected homework problems focusing on the λ-calculus from the 2016 offering of CS 4110 and from the 2017 and 2018 offerings of CS 6110. CS 4110 in 2016 consisted of 60 groups of 1–2 students, mostly undergraduates with a few masters students, while CS 6110 in both 2017 and 2018 consisted of 33 groups, mostly PhD students with some undergraduate and masters students. We omitted one illegible submission (from CS 6110 in 2017).

The problems asked questions designed to assess understanding of the λ-calculus and adjacent topics:

- Provide a translation from the call-by-value λ-calculus to call-by-name evaluation (CS 4110).
- Encode operations on Church numerals and lists (CS 4110).
- Show the reduction steps for a term under call-by-value and call-by-name evaluation (CS 6110).
- Provide examples demonstrating the importance of various side conditions in the definition of capture-avoiding substitution (CS 6110).
- Write operational semantics for full, nondeterministic β-reduction (CS 6110).
- Encode some numerical operations (CS 6110).
- Write curry and uncurry functions (CS 6110, 2017 only).
- Encode rational-number operations (CS 6110, 2018 only).

2.2 Problem Categories

We group mistakes in each solution into categories according to the conceptual misunderstanding they exhibit. This section details each conceptual category. We also categorize some mistakes as *task comprehension* errors, which are based on misunderstandings of the assignment.

We use the simplest explanation for a given error. For example, in one problem where students are asked to encode a function to check if a Church numeral is zero, a correct answer is:

$$\mathsf{ISZERO} \triangleq \lambda n.\ n\ (\lambda x.\ \mathsf{FALSE})\ \mathsf{TRUE}$$

[1] Site: https://capra.cs.cornell.edu/lambdalab/
Source: https://github.com/cucapra/lambdalab

And one incorrect answer is:

$$\mathsf{ISZERO} \triangleq \lambda n.\ n\ \lambda x.\ \mathsf{FALSE}\ \mathsf{TRUE}$$

We code this mistake as *Placement of Parentheses* rather than *Encodings and Data Structures* because the answer would have been correct with a different parentheses placement.

We identify six conceptual problem categories.

Capture-Avoiding Substitution Substitution mistakes manifest as incorrect terms after a β-reduction step. These problems arise when students make mistakes applying the formal definition of capture-avoiding substitution during reduction. The underlying issue often seems to be missing intuition for the meaning-preserving property of substitution, which makes mechanical application of the rules difficult in complex terms.

Identifying Reducible Expressions An early pitfall is identifying the reducible expression, or *redex*, in a term that dictates the next β-reduction step. As terms get longer and more complex, locating the active redex can become difficult, especially if variable name shadowing is involved. As a simple example, consider the term $x\ (\lambda y.\ y)$. This term is a normal form: no more reductions can be applied. This is not always immediately evident, however; a student unfamiliar with the λ-calculus may interpret x as the argument to the abstraction $(\lambda y.\ y)$ and obtain x as the result. In the one mistake we observed in this category, a student evaluated a more complicated term incorrectly by similarly misidentifying reducible expressions.

Order of Evaluation When a student correctly identifies the redexes in a term, another difficulty arises in choosing which one to reduce next. The correct choice depends on the reduction strategy. Locating the "active" redex in this term, for example:

$$(\lambda x.\ \lambda y.\ y)\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))$$

is key to understanding the difference between call-by-name and call-by-value reduction—the evaluation order determines whether the term converges. There were two main patterns in this category: (1) encodings that work under a different evaluation strategy from the one requested, and (2) reduction sequences that picked valid redexes in the wrong order.

Placement of Parentheses When we write λ-terms in class, we use a linear sequence of symbols to depict an abstract syntax tree (AST). Parentheses and standard implicit parenthesization rules disambiguate the mapping from symbol strings to ASTs. This difference between linear and tree-structured representations can impede deeper understanding of the λ-calculus. Consider these three terms:

$$(\lambda x.\ x)\ (\lambda y.\ y) \qquad \lambda x.\ x\ \lambda y.\ y \qquad \lambda x.\ (x\ (\lambda y.\ y))$$

The latter two strings represent the same term, but the former is different and reduces differently. Because the evaluation rules say nothing about parentheses, we expect students to mentally translate between ASTs and string representations to apply the rules.

This category contains answers that would have been correct had the parentheses been placed differently. Parenthesization mistakes appear both in final answers and in intermediate steps leading to an incorrect solution.

Encodings and Data Structures In the courses we studied, students exercise their knowledge of the λ-calculus by learning how to encode higher-level constructs as λ-terms. Homework problems include functions involving Church's encodings for Booleans and natural numbers or simple data structures like lists and binary trees. These questions amount to programming exercises that test students' intuition for for the meaning of λ-calculus programs. We classified mistakes as encoding problems when the incorrect behavior was tantamount to a bug in any other programming language, indicating that the student understood the semantics of the λ-calculus but not how to program and debug with it. One such problem, for example, is an encoding of ISZERO for Church numerals that produces TRUE for $\overline{0}$ but $\lambda x.\ \mathsf{FALSE}$ on the input $\overline{1}$.

Formal Semantics The homework problems we studied include questions about operational and translational semantics for the λ-calculus. These questions can be challenging even for students who have developed a strong intuition for the λ-calculus. The problems we observed involved incorrectly creating or formalizing semantics for a language feature or evaluation strategy for the λ-calculus. They occurred exclusively in the homework problems which tasked students with producing such formal rules.

2.3 Results

Table 1 lists the number of mistakes in each category from each course's answers. In total, 66% of assignments exhibited at least one problem, excluding task comprehension issues. The most common categories are Producing Encodings (which appears at least once in 33% of assignments) and Formal Semantics (23%). Identifying Reducible Expressions was the least common category and only appeared once.

The pervasiveness of these sorts of conceptual misunderstandings in student answers indicates the potential that automated reduction can have to increase student understanding. It is difficult to attribute all of these mechanical mistakes to a simple lack of attention to detail. We posit that students are not getting enough practice with worked examples to develop the low-level mechanical skills necessary to focus on higher-level concepts. By exposing students to more worked examples and allowing them to try their own during the homework process, we can address these misunderstandings before they have the chance to reinforce themselves through repeated incorrect work.

Table 1. Number of occurrences of each problem type across each set of assignments. A single assignment can have multiple problem types.

	Conceptual Problem Categories						Total		
Course	Encoding	Evaluation	Redexes	Parentheses	Semantics	Substitution	Any	Non-Semantics	None
CS 4110 2016	23	1	1	10	20	0	40	34	20
CS 6110 2017	10	6	0	8	1	5	20	20	12
CS 6110 2018	8	5	0	4	8	7	22	19	11

3 LambdaLab

In light of the potential of automated reduction we present LambdaLab, an interactive λ-calculus programming environment designed to help students develop intuition. The primary design goal for LambdaLab is to replicate the experience of running through worked examples on the whiteboard in class or office hours. LambdaLab starts with a simple β-reducer for λ-terms and augments it with visual feedback features that imitate the ways instructors and TAs explain the examples. We focus on these advantages of in-class reduction examples:

Indicating redexes (§3.1). Instructors can point out redexes in each expression, identify which one is the next to reduce, and indicate occurrences of the variable to be substituted. This extra information can help address mistakes in the Identifying Reducible Expressions category. It can also help with Capture-Avoiding Substitution mistakes that relate to identifying the right variable and term to use in substitution.

Visualizing AST structure (§3.1). Working on the whiteboard helps students visualize the structure of terms, since instructors can group different parts of terms together when demonstrating reduction sequences. This information can help students avoid Placement of Parentheses problems.

Propose-and-verify interaction (§3.2). In-class examples are interactive: instructors can ask students to give the next step before providing the correct answer.

Multiple evaluation strategies (§3.3). Instructors will often give worked examples using a variety of evaluation strategies to help elucidate the differences between them and to address the Order of Evaluation category of problems.

Preserving encodings (§3.4). In-class examples often use "whiteboard macros," either to illustrate the concept of encodings or just to keep terms small and understandable. An instructor might preserve a human-readable name like IF, for example, through multiple reduction steps without expanding it to its definition. Or an example might show how evaluating NOT FALSE results in the term TRUE. Strategically expanding and contracting these macro definitions can help with problems in the Encodings and Data Structures category.

We developed LambdaLab with features that mimic these advantages in an online, student-guided setting. The rest of this section describes each of these features.

3.1 Visualizing Trees and Reduction

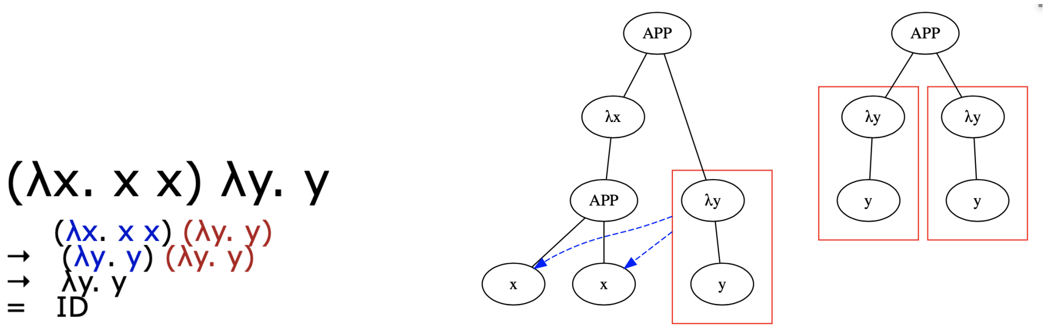
LambdaLab uses two visualization strategies to convey contextual information about terms and reductions to help students mentally connect linear strings and syntax trees. The first is a coloring scheme that indicates which parts of the term form the currently "active" redex, i.e., the next one to reduce. The coloring highlights two parts of the redex: the abstracted-over variable and its bound occurrences (blue), the expression to be substituted (red). Figure 1a shows an example reduction with this highlighting applied. The colors help students follow each reduction step by identifying which term will be substituted for which variable.

The second visualization strategy draws entire ASTs for terms in a reduction sequence using the same coloring scheme. LambdaLab can draw any pair of terms that form a β-reduction step. A student clicks on a reduction step to visualize it. Figure 1b shows an example. LambdaLab draws the pre-reduction term to the left of the post-reduction term. It adds a red box around the expression to be substituted and draws a blue arrow pointing to the variable occurrences to be substituted. The post-reduction term keeps the substituted expression boxed in red to emphasize that it has "moved" to the new locations in the term.

Together, these two visualization features recreate the effect of an instructor walking through the process of a β-reduction pointing out the active redex and the parts of the term that are involved in the reduction and strategically drawing ASTs to resolve parsing ambiguities.

3.2 Interactive Mode

In the default mode, when a user types a term into the input box, LambdaLab immediately displays the entire reduction sequence (up to a timeout). This eager evaluation, however, can "spoil" the answers to questions about β-reduction. During an interactive demonstration, an instructor can ask for suggestions for the term that results from reduction and give feedback about its correctness. To replicate this instructional strategy, LambdaLab has an *interactive mode* where the student proposes potential reduction results.

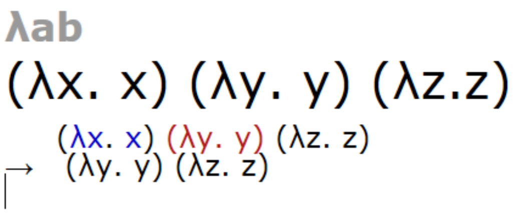


```
(λx. x x) λy. y
    (λx. x x) (λy. y)
→   (λy. y) (λy. y)
→   λy. y
=   ID
```

(a) Term entry and reduction sequence.

(b) Reduction step visualization.

Figure 1. The main LambdaLab interface showing the evaluation of a simple term. The tree shows the β-reduction step; the red-boxed area on the left replaces the variable occurrences that the blue arrow points to.



```
λab
(λx. x) (λy. y) (λz.z)
    (λx. x) (λy. y) (λz. z)
→   (λy. y) (λz. z)
```

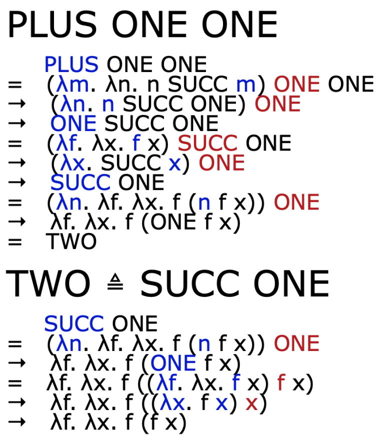
Figure 2. Partially complete evaluation of a term in interactive mode. Students enter the next part of the evaluation sequence at the cursor. (This user interface is not finalized.)

Figure 2 depicts the interactive mode. The user types a proposed result term, and LambdaLab checks whether an α-equivalent term exists *anywhere* in the reduction sequence for the original input term. That is, students need not type the *next* step in the reduction sequence—they may "jump ahead" to any point in the reduction. LambdaLab fills in the gap with the subsequence of terms up to the point of the correct term. The user can continue iterating through the reduction sequence by typing new, further-reduced terms. Through this process, the student collaborates with LambdaLab to advance through the complete reduction sequence until no more steps remain.

If the student ever enters a term that does not exist in the reduction sequence, LambdaLab rejects the term and does fill any steps. We view this rejection as an important avenue for future work: a future version of LambdaLab should automatically generate feedback to indicate what went wrong and to help guide the student toward a correct term.

3.3 Evaluation Strategies

LambdaLab can switch its evaluator between four common reduction strategies: Call-By-Name, Call-By-Value, Normal Order, and Applicative Order. Students can switch between the four strategies at any time, and evaluation steps for the entered term will be displayed immediately. This allows students to quickly compare and contrast the result of evaluating a term under the different strategies.



```
PLUS ONE ONE
    PLUS ONE ONE
=   (λm. λn. n SUCC m) ONE ONE
→   (λn. n SUCC ONE) ONE
→   ONE SUCC ONE
=   (λf. λx. f x) SUCC ONE
→   (λx. SUCC x) ONE
→   SUCC ONE
=   (λn. λf. λx. f (n f x)) ONE
→   λf. λx. f (ONE f x)
=   TWO
TWO ≜ SUCC ONE
    SUCC ONE
=   (λn. λf. λx. f (n f x)) ONE
→   λf. λx. f (ONE f x)
=   λf. λx. f ((λf. λx. f x) f x)
→   λf. λx. f ((λx. f x) x)
→   λf. λx. f (f x)
```

Figure 3. Term entry (above) and macro definition (below) in the LambdaLab interface. In each part, LambdaLab marks β-reduction steps with ⟶ and macro expansion or contraction with =.

3.4 Encoding Expansion and Contraction

Many realistic examples of λ-calculus evaluation involve an informal notion of *macros*, which are words we write to stand in for longer terms. During a demonstration of the Church encodings for Booleans, for example, we might write AND FALSE TRUE and reduction steps eventually leading to FALSE. Students need to understand that words like AND and TRUE are not actually part of the λ-calculus syntax and are not visible to its semantics rules: they are an informal construct that saves time and space, but they are only formally meaningful once they are desugared into plain λ-terms. An important challenge in designing LambdaLab is automating the way that instructors informally treat these macros to make examples more intelligible.

Users enter macro definitions separately from terms to be evaluated. Figure 3 depicts the term and macro-definition entry views in LambdaLab. In this example, the user has already provided definitions for SUCC, the successor function on Church numerals, and the number ONE; they can use these encodings to define the new macro TWO. (Macros must use all-capital names to distinguish themselves from variables.) LambdaLab detects cycles in macro definitions to prevent macros that (directly or indirectly) reference themselves. It issues an error that helps motivate the need for fixed-point combinators in λ-calculus programming.

LambdaLab preserves macro names through reduction when possible, but it selectively eliminates macro names when substitution occurs "inside" the expanded term. In Figure 3, each line is either a plain β-reduction step, marked with a →, or an equality, marked with an =, in which a single macro is substituted. Crucially, LambdaLab both expands and *contracts* macros, eliminating longer terms and substituting them for short names. The last line of the top reduction, for example, is an equality step showing a term being rewritten to TWO. Macro contraction is important for emphasizing when the results of an encoded computation are correct. In interactive mode (Section 3.2), users may enter terms with macros either expanded or contracted—LambdaLab considers them equivalent.

LambdaLab's macro support raises questions about term equivalence and when to expand and contract macros, which we address in the next section.

4 λ-Calculus with Macros

The behavior of macros in LambdaLab is surprisingly subtle. We needed to resolve a variety of questions about when macros expand and contract during reduction and how to stay agnostic to the chosen evaluation order. These questions do not arise in the informal setting of an in-class worked example, but they are critical to automating a process that closely resembles the that informal behavior. This section describes our formalization that answers these questions.

We describe a formal language, the λ-Calculus with Macros, that extends the λ-calculus with explicit macro terms. We design its semantics to match the intuition behind in-class reductions as closely as possible—while acknowledging that it may be impossible to exactly replicate the nondeterministic choices instructors make. In this language, terms can take both ordinary β-reduction steps and also *equality steps*, wherein the term expands or contracts macro definitions to produce an equivalent term. We build on two core design principles:

1. Macros should be expanded as late as possible without obscuring the β-reduction steps that involve them.
2. Reduction should be unaffected by macros. A term containing macro references should take the same β-reduction steps as one where the names are replaced with their equivalent terms.

For example, consider this term with macro references:

$$(\lambda x.\ x\ \Omega)\ \mathsf{FALSE}$$

where FALSE is an encoding for $\lambda x.\ \lambda y.\ y$ and Ω is the standard paradoxical combinator, $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$. Under call-by-name evaluation, this term converges in two reduction steps:

$$\begin{aligned} &(\lambda x.\ x\ \Omega)\ \mathsf{FALSE} \\ \to\ &\mathsf{FALSE}\ \Omega \\ =\ &(\lambda x.\ \lambda y.\ y)\ \Omega \\ \to\ &\lambda y.\ y \end{aligned}$$

wherein the third line shows a macro expansion to reveal the next redex in CBN order.

Under call-by-value evaluation, this term diverges. A naïve application of our first principle, however, would only expand the left-hand side of applications and leave Ω in $\mathsf{FALSE}\ \Omega$ unexpanded—yielding the same answer as the CBN semantics. To adhere to the second principle, CBV and CBN need different macro expansion rules.

Other reduction orders, including normal and applicative order, similarly require special rules about when to expand macros, although for different reasons. These two strategies must deal with the open terms that occur inside of abstractions, in which free variables can appear on the left-hand sides of applications, while CBV and CBN need not consider anything inside an abstraction until it is applied. In this sense, the point at which macro expansion becomes "necessary" differs from the two simpler strategies.

4.1 Syntax and Semantics

This section develops the semantics for the λ-Calculus with Macros for each evaluation order. We use a syntax that extends the λ-calculus with one new form:

$$e ::= x \mid \lambda x.\ e \mid e_1\ e_2 \mid M$$

where M is a metavariable ranging over a countable set of macro names. We will also use the metavariable σ to denote mappings from macro names to expressions, which hold the current macro definitions.

We define a reduction judgment $\sigma \vdash e_1 \overset{\square}{\to} e_2$ for each evaluation order, where $\square \in \{\to, =\}$. The symbol above the arrow distinguishes standard β-reduction steps ($\to$) from our equality steps (=). In an equality step $\sigma \vdash e_1 \overset{=}{\to} e_2$, the terms e_1 and e_2 are equivalent modulo macro expansion from σ.

Figure 4 lists the rules for each evaluation order. The rules for reduction steps ($\overset{\to}{\to}$) reflect traditional small-step rules for the λ-calculus, but the equality-step rules ($\overset{=}{\to}$) are new.

$$\frac{\sigma \vdash e_1 \overset{\square}{\rightarrow} e_1'}{\sigma \vdash e_1\; e_2 \overset{\square}{\rightarrow} e_1'\; e_2} \qquad \frac{}{\sigma \vdash (\lambda x.\; e_1)\; e_2 \overset{\rightarrow}{\rightarrow} e_1\{e_2/x\}}$$

$$\frac{}{\sigma \vdash M\; e \overset{=}{\rightarrow} \sigma[M]\; e}$$

(a) Call-by-name.

$$\frac{\sigma \vdash e_1\; e_2 \overset{\square}{\rightarrow} e'}{\sigma \vdash (e_1\; e_2)\; e \overset{\square}{\rightarrow} e'\; e} \qquad \frac{}{\sigma \vdash (\lambda x.\; e_1)\; e_2 \overset{\rightarrow}{\rightarrow} e_1\{e_2/x\}}$$

$$\frac{\sigma \vdash e \overset{\square}{\rightarrow} e'}{\sigma \vdash \lambda x.\; e \overset{\square}{\rightarrow} \lambda x.\; e'} \qquad \frac{\sigma \vdash e \overset{\square}{\rightarrow} e'}{\sigma \vdash w\; e \overset{\square}{\rightarrow} w\; e'}$$

$$\frac{}{\sigma \vdash M\; e \overset{=}{\rightarrow} \sigma[M]\; e} \qquad \frac{\sigma \vdash \sigma[M] \overset{\rightarrow}{\rightarrow} e}{\sigma \vdash w\; M \overset{=}{\rightarrow} w\; \sigma[M]}$$

$$v ::= \lambda x.\; v \mid w$$
$$w ::= x \mid w\; v$$

(b) Normal order.

$$\frac{\sigma \vdash e_1 \overset{\square}{\rightarrow} e_1'}{\sigma \vdash e_1\; e_2 \overset{\square}{\rightarrow} e_1'\; e_2} \qquad \frac{}{\sigma \vdash (\lambda x.\; v)\; e \overset{\rightarrow}{\rightarrow} v\{e/x\}}$$

$$\frac{\sigma \vdash e \overset{\square}{\rightarrow} e'}{\sigma \vdash \lambda x.\; e \overset{\square}{\rightarrow} \lambda x.\; e'} \qquad \frac{\sigma \vdash e \overset{\square}{\rightarrow} e'}{\sigma \vdash w\; e \overset{\square}{\rightarrow} w\; e'}$$

$$\frac{}{\sigma \vdash M\; e \overset{=}{\rightarrow} \sigma[M]\; e} \qquad \frac{\sigma \vdash \sigma[M] \overset{\rightarrow}{\rightarrow} e}{\sigma \vdash w\; M \overset{=}{\rightarrow} w\; \sigma[M]}$$

$$v ::= \lambda x.\; v \mid w$$
$$w ::= x \mid w\; v$$

(c) Applicative order.

$$\frac{\sigma \vdash e_1 \overset{\square}{\rightarrow} e_1'}{\sigma \vdash e_1\; e_2 \overset{\square}{\rightarrow} e_1'\; e_2} \qquad \frac{\sigma \vdash e \overset{\square}{\rightarrow} e'}{\sigma \vdash v\; e \overset{\square}{\rightarrow} v\; e'}$$

$$\frac{}{\sigma \vdash (\lambda x.\; e)\; v \overset{\rightarrow}{\rightarrow} e\{v/x\}} \qquad \frac{}{\sigma \vdash M\; e \overset{=}{\rightarrow} \sigma[M]\; e}$$

$$\frac{\sigma \vdash \sigma[M] \overset{\rightarrow}{\rightarrow} e}{\sigma \vdash v\; M \overset{=}{\rightarrow} v\; \sigma[M]}$$

$$v ::= \lambda x.\; e$$

(d) Call-by-value.

Figure 4. Small-step operational semantics for the λ-Calculus with Macros under four evaluation orders.

The simplest rule appears in the call-by-name semantics (Figure 4a): it expands any macro that appears on the left-hand side of an application. In CBN, macros on the right-hand side never expand. The other three orders require a notion of fully-evaluated terms (i.e., values) to control expansion. In normal and applicative order, we define metavariables v and w encompassing terms that cannot step. Each w term begins with a free variable, and a value v consists of abstractions wrapping such a w term. In these orders, macros may expand on the right-hand side of applications when the left-hand side is such a w.

We include premises to avoid needlessly expanding macros that cannot reduce. For example, the final rule in the call-by-value semantics (Figure 4d) requires that $\sigma[M]$ can take a reduction step. This rule helps distinguish between cases like our above example, FALSE Ω, and ones where the right-hand side is a value, as in NOT TRUE. The former term should expand its right-hand macro to ensure correct evaluation, while the latter should expand its left-hand macro to preserve the term's simplicity. The equality rule "peeks" inside macro definition on the right-hand side of an application to check whether expanding it allows a step.

Each semantics includes "propagation" rules that let steps occur in nested subexpressions (e.g., the first call-by-name rule in Figure 4a). These rules use the metavariable □ to range over step types, so both equality and reduction steps can propagate in the same way.

4.2 Macro Equivalence and Contraction

Realistic λ-calculus examples also *contract* macro definitions. A reduction sequence for SUCC ONE as in Figure 3, for example, should eventually end in TWO if its definition is available. LambdaLab needs to decide when to contract a subterm to rewrite it as a macro name.

A naïve strategy would contract any subterm that is observationally equivalent to a macro definition. In addition to being undecidable, however, observational equivalence fails to distinguish between any two diverging terms. If OMEGA is defined as $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$, for example, it would be confusing for LambdaLab to unconditionally rewrite any diverging term as OMEGA.

To address the latter problem, we consider an idealized (but still undecidable) equivalence relation $\equiv_*$ that distinguishes between terms that "diverge differently":

$$\sigma \vdash e_1 \equiv_* e_2 \iff \exists e'.\ e_1 \to_\beta^* e' \land e_2 \to_\beta^* e'$$

Terms equivalent under $\equiv_*$ are also observationally equivalent, but the converse does not hold. The rest of this section develoHps decidable approximations of $\equiv_*$.

Macro expansion. We first formalize a simple translation for syntactically expanding all macros in a term:

$$\begin{aligned} \sigma[\![x]\!] &\triangleq x \\ \sigma[\![\lambda x.e]\!] &\triangleq \lambda x.[\![e]\!] \\ \sigma[\![e_1 e_2]\!] &\triangleq \sigma[\![e_1]\!]\sigma[\![e_2]\!] \\ \sigma[\![M]\!] &\triangleq \sigma[M] \end{aligned}$$

LambdaLab requires macro definitions to be closed, so variable capture is not a problem here.

Normal-order equivalence. We define two terms to be equivalent if their results under normal-order reduction are α-equivalent:

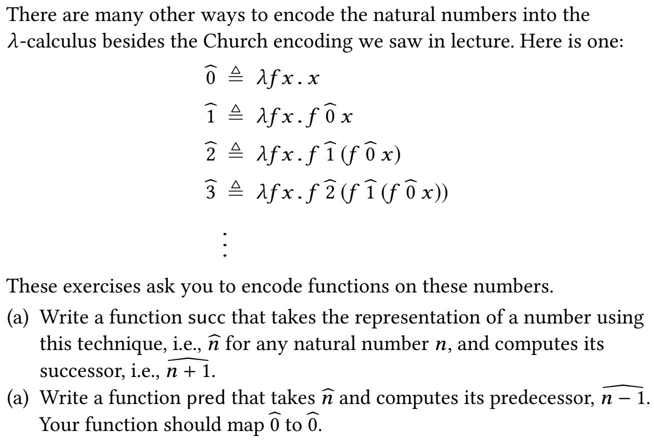
$$\begin{aligned} \sigma \vdash e_1 \equiv_{\text{norm}} e_2 \iff & \sigma[\![e_1]\!] \Downarrow_{\text{norm}} v_1 \land \sigma[\![e_2]\!] \Downarrow_{\text{norm}} v_2 \\ & \land\ v_1 =_\alpha v_2 \end{aligned}$$

Normal-order reduction is guaranteed to find a normal form for a term if one exists, so $\equiv_{\text{norm}}$ suffices for terms that converge. Divergent terms are never considered equivalent, even to themselves, so LambdaLab does not collapse them.

Order-specific equivalence. The choice of evaluation order, however, can influence the appropriate definition of equivalence. Specifically, two terms may evaluate under a given order to the same term even if they do not have a normal form. Recursive functions defined using fixed-point combinators, for example, can have no normal form even when they quickly evaluate to CBV or CBN values. We can define a family of relations $\equiv_s$, where s is the strategy in question:

$$\begin{aligned} \sigma \vdash e_1 \equiv_s e_2 \iff & \sigma \vdash e_1 \equiv_{\text{norm}} e_2 \\ & \lor (\sigma[\![e_1]\!] \Downarrow_s v_2 \land \sigma[\![e_2]\!] \Downarrow_s v_2 \land v_1 =_\alpha v_2) \end{aligned}$$

These relations consider terms equivalent if they have a common normal form *or* converge to an order-specific common value. This additional flexibility finds some equivalent terms that happen to produce different values under a given evaluation order. Consider, for example, $\lambda f.\ \lambda x.\ f$ (ONE $f\ x$) and $\lambda f.\ \lambda x.\ f\ (f\ x)$, which reduce differently under CBV but the same way under normal order, and are both equivalent to TWO.


There are many other ways to encode the natural numbers into the λ-calculus besides the Church encoding we saw in lecture. Here is one:

$$\begin{aligned} \widehat{0} &\triangleq \lambda f x.\, x \\ \widehat{1} &\triangleq \lambda f x.\, f\, \widehat{0}\, x \\ \widehat{2} &\triangleq \lambda f x.\, f\, \widehat{1}\, (f\, \widehat{0}\, x) \\ \widehat{3} &\triangleq \lambda f x.\, f\, \widehat{2}\, (f\, \widehat{1}\, (f\, \widehat{0}\, x)) \\ &\vdots \end{aligned}$$

These exercises ask you to encode functions on these numbers.

(a) Write a function succ that takes the representation of a number using this technique, i.e., $\widehat{n}$ for any natural number n, and computes its successor, i.e., $\widehat{n+1}$.

(a) Write a function pred that takes $\widehat{n}$ and computes its predecessor, $\widehat{n-1}$. Your function should map $\widehat{0}$ to $\widehat{0}$.

Figure 5. A question on encoding natural numbers.

Furthermore, equivalence under any $\equiv_s$ implies $\equiv_*$. LambdaLab implements each $\equiv_s$ as a reasonably reliable approximation of $\equiv_*$ to decide when to replace subterms with macro names. To ensure that it collapses the largest possible subterms, the contraction algorithm walks ASTs in breadth-first order and tests each for $\equiv_s$-equivalence with each macro. When it finds the first (largest) equivalence, LambdaLab inserts an equality step that contracts the macro.

5 Evaluation and Case Studies

This feature set helps LambdaLab address the underlying misunderstandings for five of the problem types we observed during our empirical investigation (Section 2.2): Capture-Avoiding Substitution, Identifying Reducible Expressions, Order of Evaluation, Encodings and Data Structures, and Placement of Parentheses. Out of the 125 assignment submissions we studied, 73 (58%) exhibit at least one of these five problems (see Table 1). When we exclude the assignments with only Task Comprehension problems (or no problems), 89% of the remaining submissions include at least one problem category that LambdaLab addresses. While we find that LambdaLab targets the right problem categories for many students, we also see the prevalent Formal Semantics category as an avenue for future work.

To investigate LambdaLab's potential in more detail, this section uses case studies that examine specific homework answers. We describe concrete ways that the student might use LambdaLab to identify and rectify inaccuracies in their understanding.

Debugging Encodings Figure 5 shows a CS 6110 problem that asks students to encode a predecessor function using a non-standard encoding of the natural numbers. A simple correct answer is:

$$\text{PRED} \triangleq \lambda n.\ n\ (\lambda x.\ \lambda y.\ x)\ \widehat{0}$$

Consider the closed λ-term $(\lambda x\,.\,\lambda y\,.\,x\;y)\,((\lambda x\,.\,\lambda z\,.\,x\;z)\,(\lambda y\,.\,y))$.

(a) Fully reduce the term using *call-by-value* (CBV) evaluation. Show each step in the β-reduction.

(a) Fully reduce the term in *call-by-name* (CBN) order, again showing each step.

Figure 6. A question on evaluation strategies.

One student gave this incorrect answer:

$$\mathsf{PRED} \triangleq \lambda n.\ n\ (\lambda x.\ \lambda y.\ x)\ (\lambda x.\ x)$$

which is correct for all inputs except zero, where it produces the identify function instead of zero. This is the kind of simple mistake that an automated reducer can help catch. With access to LambdaLab, students can enter their encoding for PRED and a few example numbers and evaluate several test inputs such as PRED ONE and PRED ZERO to check the result. LambdaLab's macro contraction serves as a verifier: the last line in the reduction for PRED ZERO should be rewritten to ZERO. This incorrect solution would instead reduce to ID, a built-in macro for the identity function. The student could then inspect the reduction sequence to find where it went wrong and try a new value for PRED.

This process mimics the way that real-world programmers use test cases to identify and fix bugs. One potential criticism is that this debugging process lets students brute-force solutions by blindly searching for a term that works. This approach, however, is already possible—although more tedious—with manual reduction. LambdaLab's automation avoids focusing penalties on mechanical mistakes instead of conceptual encoding errors.

Comparing Evaluation Strategies Figure 6 shows a problem that asks students to evaluate the same term under both call-by-value and call-by-name reduction. The call-by-value result is:

$$\lambda y.\ (\lambda z.\ (\lambda y.\ y)\ z)\ y$$

and the call-by-name result is:

$$\lambda y.\ ((\lambda x.\ \lambda z.\ x\ z)\ (\lambda y.\ y))\ y$$

A common mistake in our data set is to reduce both the same way. This mistake indicates a lack of understanding of the difference between the evaluation strategies. Using LambdaLab, students can enter a term and switch between CBV and CBN evaluation to see where they differ. If they are confused about how a particular reduction step worked, they can click on it to show a tree view that highlights the active terms in that redex. They can also modify the term to immediately see the effects on the evaluation sequence.

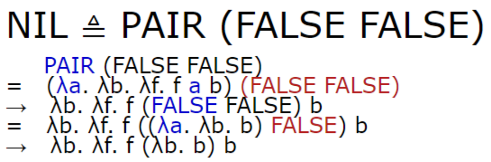
This capability of LambdaLab also highlights a problem in the homework question itself. Mechanical reduction problems are unlikely to teach students anything beyond the mechanical application of semantics rules. The problem might be improved by making it more conceptual: it could ask students to explain the root causes of the different results and to point out which parts of the semantic rules give rise to the differing behavior.

Define expressions for each of these list constructs:

- NIL, the empty list;
- CONS, a function that adds an element to the head of a list and returns a new list;
- HD, a function that gets the head element in a list;
- TL, a function that gets the tail of a list (i.e., the input list without its head element); and
- IS_NIL, a function that checks whether a list is empty (and returns a Church boolean).

To keep things simple, applying either HD or TL to NIL should produce NIL. Also, if you find it helpful, you may use the encodings of Booleans and pairs discussed in class.

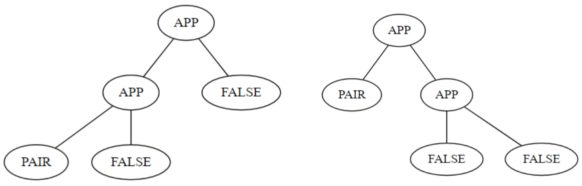
Figure 7. A homework problem on encoding lists.



```
NIL ≜ PAIR (FALSE FALSE)

     PAIR (FALSE FALSE)
=    (λa. λb. λf. f a b) (FALSE FALSE)
→    λb. λf. f (FALSE FALSE) b
=    λb. λf. f ((λa. λb. b) FALSE) b
→    λb. λf. f (λb. b) b
```

Figure 8. LambdaLab's output on an incorrect encoding for the empty list.



(a) Correct encoding. **(b)** Incorrect encoding.

Figure 9. Tree visualizations for two encodings of NIL highlighting the importance of parenthesis placement.

Parenthesizing Expressions Figure 7 shows a CS 4110 assignment that asks students to design an encoding for lists. It is open-ended: students can choose any encoding strategy. A common choice uses Lisp-style cons cells tagged with a Boolean: a list is a pair of a Boolean indicating whether it is empty and, for non-empty lists, another pair containing the head and the tail.

One such answer included a correct pair constructor:

$$\mathsf{PAIR} \triangleq \lambda a.\ \lambda b.\ \lambda f.\ f\ a\ b$$

But gave an incorrect encoding for the empty list:

$$\mathsf{NIL} \triangleq \mathsf{PAIR}\ (\mathsf{FALSE}\ \mathsf{FALSE})$$

A correct encoding is only slightly different:

$$\mathsf{NIL} \triangleq \mathsf{PAIR}\ \mathsf{FALSE}\ \mathsf{FALSE}$$

because the second element in the pair does not matter when the first is FALSE. The placement of the parentheses means that NIL is not a proper pair value.

Figure 8 shows LambdaLab's interface when defining this incorrect encoding. The simplified result indicates that something has gone wrong: the term is not the pair encoding $\lambda f.\ f$ FALSE FALSE as expected. The reduction sequence can show where things went awry: LambdaLab highlights the entire term (FALSE FALSE) in red to show that it is an argument to the pair function. Figure 9 shows LambdaLab's tree output for the two different answers, further highlighting the difference in the structure of the two terms.

6 Related Work

As an advanced topic in the computer science curriculum, the λ-calculus has received less attention from pedagogical research than broader, more introductory topics. Sestoft describes an early web interface for experimenting with reduction strategies [9], but the system lacks interactive features: it just displays reduction sequences. (The site is also no longer accessible at the time of submission.) Rojas's λ-calculus tutorial [8] includes a color-coding visualization that inspired the way that LambdaLab automatically indicates the expressions involved in each reduction step. Buliga designs a language for visualizing the λ-calculus using diagrams [2]. The graph language is part of a larger logical formalization, and it is unclear whether this style of visualization would be useful for beginners. The most closely related tool is Jhala's Elsa [6], which focuses on teaching and building intuition about the λ-calculus by verifying reduction sequences that students propose. It works as a proof checker that checks whether a given reduction sequence follows from the evaluation rules. LambdaLab's interactive mode uses the same propose-and-verify approach but allows students to skip some reduction steps to focus on converging on the final reduction result.

Pombrio and Krishnamurthi's work on *resugaring* [7] informed our macro contraction process (Section 4.2). Their paper includes a proof of equivariance: that terms are α-equivalent in the sugared language if they are α-equivalent in the desugared language. We ensure this property in LambdaLab along with the semantic equivalence property.

Work on multi-stage programming [10] and meta-λ-calculi [1, 11] inspired our macro system and its formalization. Our λ-Calculus with Macros is simpler because our macros must be closed terms, so cross-level references are not allowed. Unlike that work, our formalization describes when and how to expand macros to make reduction sequences understandable.

There has also been a profusion of work on explaining the reduction semantics of languages other than the λ-calculus. Clements, Flatt and Felleisen describe a tool [4] for algebraically rewriting programs in Scheme which operates similarly to our method in LambdaLab. However, the work is concerned more with the tool's applications to debugging than its potential use in education. A subsequent Racket-embedded DSL called PLT Redex [5] is more pedagogically oriented, but the language and its accompanying book are more interested in teaching the general concepts of programming language semantics than in the λ-calculus specifically.

7 Conclusion

In a recent offering of CS 6110, anecdotal student response to LambdaLab as a supplementary tool has been positive—and it has illuminated areas for improvement. The clear next step is to integrate LambdaLab more deeply into the design of homework problems and to perform a rigorous user study of its impact on learning. LambdaLab's user interface is still primitive, so principled work on its design will aid adoption.

Aside from LambdaLab itself, our empirical investigation into student mistakes raises questions about the aims of teaching the λ-calculus. What do we hope students will gain from learning about it? Is β-reduction intuition valuable for students not intending to study programming languages past an undergraduate level? How is teaching the λ-calculus similar to, and different from, teaching a "real" programming language? With the major categories of mistakes in mind, what are the best pedagogical strategies to address them? While an interactive tool can help, opportunity abounds to apply educational techniques to this obstacle in the programming languages curriculum.

Acknowledgments

Many thanks to the CS 4110 and 6110 students who used LambdaLab and gave feedback while it was under development. Even more thanks to Molly Q Feldman, who guided the authors through the evaluation process and through their first interaction with Cornell's institutional review board.

References

[1] Daisuke Bekki. 2008. Monads and meta-lambda calculus. In *Annual Conference of the Japanese Society for Artificial Intelligence*. Springer, 193–208.

[2] Marius Buliga. 2013. Graphic lambda calculus. *Complex Systems* 22, 4 (2013), 311–360.

[3] Alonzo Church. 1936. A note on the Entscheidungsproblem. *The journal of symbolic logic* 1, 1 (1936), 40–41.

[4] John Clements, Matthew Flatt, and Matthias Felleisen. 2001. Modeling an algebraic stepper. In *European symposium on programming*. Springer, 320–334.

[5] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics engineering with PLT Redex*. Mit Press.

[6] Ranjit Jhala. 2018. Elsa. https://github.com/ucsd-progsys/elsa

[7] Justin Pombrio and Shriram Krishnamurthi. 2014. Resugaring: lifting evaluation sequences through syntactic sugar. In *Programming Language Design and Implementation (PLDI)*.

[8] Raúl Rojas. 2015. A tutorial introduction to the lambda calculus. *arXiv preprint arXiv:1503.09060* (2015).

[9] Peter Sestoft. 2002. Demonstrating Lambda Calculus Reduction. In *The essence of computation*. Springer, 420–435.

[10] Walid Taha and Tim Sheard. 1997. Multi-stage Programming with Explicit Annotations. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*.

[11] Kazunori Tobisawa. 2015. A Meta Lambda Calculus with Cross-Level Computation. In *Principles of Programming Languages (POPL)*.