Towards a Notional Machine for Runtime Stacks and Scope: When Stacks Don't Stack Up

John Clements*
clements@calpoly.edu
Cal Poly
San Luis Obispo, CA, USA

Shriram Krishnamurthi
shriram@brown.edu
Brown University
Providence, RI, USA

ABSTRACT

Background and Context. Modern programming is replete with features like closures, callbacks, generators, asynchronous functions, and so on. These features can be subtle in their behavior and interaction with the rest of the language, so students need notional machines that are both accurate and manageable. We specifically focus on stacks and their associated environments, which are key to understanding these features.

Objectives. What conceptions do students have of stacks and environments on entry to a tertiary, upper-level programming languages course? What impact does coursework around an interpreter-based approach to programming languages education have on their understanding? What is the value of the tooling we create for students to express notional machine states?

Method. Our studies were conducted at two different institutions in a tertiary, upper-level course on programming languages. The intermediate interventions were different, but both used a pre-/post-test format conducted at the beginning and near the end of the semester. We also created a Snap*!*-based tool to assist in notional machine description.

Findings. We found that students have a relatively weak understanding of stacks and environments at entry, and exhibited various misconceptions. At one institution, which primarily relied on interpreters, we found numerous problems persisted at the end of the semester. At the other, which in addition to interpreters also used direct instruction and the Snap*!*-based tool, students did much better—but still not on aspects that we would have liked to have transferred from interpreters.

Implications. Our findings suggest that it is important for other educators to also assess their students' understanding of stacks and environments, especially in light of modern programming concepts. Assuming similar misconceptions are widespread, we believe the community needs to invest much more effort into notional machines, so that students can better understand the features and programs they are working with. In particular, the "standard" stack presentation is insufficient and in some cases even "harmful" (since it reinforces some misconceptions), and hence must be revised. Finally, our study forces a reconsideration of the learning objectives met by interpreters in programming languages education.

*Work was conducted while on sabbatical at Brown University.

ICER 2022, August 7–11, 2022, Lugano and Virtual Event, Switzerland
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9194-8/22/08…$15.00
https://doi.org/10.1145/3501385.3543961

CCS CONCEPTS

• **Applied computing → Education**; • **Software and its engineering → General programming languages**.

KEYWORDS

stacks, environments, misconceptions, scope, control

ACM Reference Format:
John Clements and Shriram Krishnamurthi. 2022. 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 V.1 (ICER 2022), August 7–11, 2022, Lugano and Virtual Event, Switzerland*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3501385.3543961

1 INTRODUCTION

Modern programming uses a variety of rich control features, such as generators, threads, callbacks, exceptions, coroutines, recursion, and so on. These, in turn, can depend on first-class functions, such as the `lambda` construct (henceforth, *closures*) of many languages. Some of these can appear early in a programming curriculum, e.g., if a student programs GUIs, asynchronous input-output, or with higher-order functions.

To make sense of these constructs, students need an accurate notional machine. A key part of a notional machine for these kinds of constructs is an accurate depiction of the stack, and of its related name bindings (the mapping of variables to values). The premise of this paper is that the traditional depiction of the stack, as a sequence of self-contained frames, is insufficient and potentially misleading, mirroring and even creating troubling misconceptions. We investigate this in the context of courses on the principles of programming languages, with students who already have extensive programming background and some awareness of the stack from prior systems programming courses.

Concretely, we address the following research questions:

(1) What understanding do students have of stacks at entry into a tertiary, upper-level programming languages course?
(2) What problems remain after a term of instruction? In particular, is the interpreter-based approach (see section 2) effective at addressing their initial misconceptions?
(3) What value do we find in the tooling we created for students to depict stacks, environments, and heaps?

2 THEORIES OF INSTRUCTION IN PROGRAMMING LANGUAGES

There are many approaches to teaching programming languages. Here we discuss those most relevant to this paper.

One approach, popularized by a number of widely-used books [1, 12, 16, 17], is based on *definitional interpreters.* Following the tradition of McCarthy's [20] seminal paper, students write interpreters for the essence of programming languages. This approach has the virtue that the *differences* between languages can often be captured very concisely: e.g., only a few lines of code distinguish eager and lazy evaluation, and students can focus on the consequences of these lines. This approach is constructionist (and hence constructivist), because it believes there is value to students implementing languages for themselves, giving them a medium with which to experiment with language designs.

In the specific courses we study, students implement interpreters in Racket [8], a descendant of Scheme, again following a long tradition [1, 12]. Concretely, the courses both follow the textbook *Programming Languages: Application and Interpretation* [17], which is used at more than 50 institutions.[1]

Another approach is the use of *mystery languages* (ML) [4, 24]. Here, students explore comparative linguistics by writing programs in one syntax that is fed to multiple different implementations, each of which may produce different answers. These answers represent different language designs. For instance, when exploring different designs for arithmetic, `1 + 2` may return `3` in all the implementations, but `1 / 0` may produce an error in one, `NaN` in a second, and `0` in a third, reflecting what different languages have done. In this approach, students are expected to construct programs to tell apart the languages, using a version of the "scientific method": they explore until they hit on a difference, hypothesize the underlying theory, derive consequences from the theory, generate new programs to explore those consequences, and run them to confirm their understanding.

These approaches are complementary. Interpreters focus on implementations, leaving the consequential behaviors for students to derive. MLs focus on behaviors, leaving the implementations for students to derive. In both cases, students actively engage in constructing meaning through writing code. Traditionally, however, as in our courses, interpreter approaches give students a fair bit of initial instruction (with additional exploration found in writing strong *test suites*!), while ML approaches tend to be more purely exploratory, with minimal hints on how students should explore.

Another theory of instruction, not just for understanding languages but also for programming itself, is that notional machines [5] are a useful and perhaps even necessary tool for program comprehension.[2] Being a kind of programming language semantics [18], they connect directly to a programming languages course. We believe notional machines have been discussed in enough detail in computing education that, given space constraints, we do not need to belabor the point here; we refer readers to Sorva's survey [27] and dissertation [26]. We simply note that we adhere to this theory, and hence focus on the analysis of existing notional machines and make progress towards a new one.

Terminology. There does not seem to be a canonical order for stack growth. In our presentation, stacks grow "downward". However, nothing in our work depends on this, and indeed we did not observe (see section 7 and section 9) any problems in student work that could be traced to the stack growing in a direction opposite what they may have been taught earlier. To avoid confusion in this paper, we do not refer to the spatial position of frames ("above" and "below", for instance), but rather to their *temporal* (age) ordering, i.e., "earlier" or "older" and "later" or "newer" frames. Visually, older frames will appear above and newer frames below.

3 RELATED WORK

Sorva's works [26, 27] provide an extensive discussion of notional machines and visualizations. It is notable that most of these do not cover features like closures and threads and/or have the problems that we discuss in section 4.

The Python Tutor [14] is widely used. However, in our estimation, its conflation of stack frames and environments leads to non-stack-like popping of intermediate frames; its handling of unbound identifiers may or may not resolve scope misconceptions; and its line-orientedness and lack of context information (see section 8) makes it a poor fit for programs with fine-grained returns. Finally, it does not handle control features like threads, concurrency, and asynchrony at all (which, again, would benefit from context information). We have looked for formal evaluations of its visualization. Unfortunately, neither our searches nor the references cited by Guo [13] yield papers that focus on the features we cover in this paper: the only usability evaluations are for rudimentary programs, for which Python Tutor may be well-suited.

Other authors have proposed visualization systems that do better:

- Sorva's UUhistle [26] uses evaluation-context-like context descriptions (see section 8), and has been tested for usability. However, it may suffer from suggesting dynamic scope due to the placement of frames. We do not believe this aspect of it has been evaluated. It also does not support more advanced control like generators. A recent paper discusses many more tools that are expression- rather than line-oriented [6, section 6.2.6], but most of these lack evaluation, especially of the aspects we focus on.
- Pollock, et al. [23] present Theia, a framework to *generate* notional machine visualizations. Their paper echoes some of our criticisms of visualizations in Python Tutor (e.g., in the handling of closures and suggestion of dynamic scope). However, the paper does not include any evaluation of Theia.
- *How to Design Programs* [7] provides a correct and complete *series* (based on language levels [11]) of notional machines using substitution. This is also implemented by the Racket Algebraic Stepper [3]. However, we have encountered several usability problems with this tool. Unfortunately, we are not aware of any usability evaluation of the tool at all. Our contexts (see section 8) are inspired by the Stepper but also attempt to fix the problems that we have seen.

[1]Information provided by the author.
[2]We specifically use the term to mean a mechanical description of system behavior, not a mental model in the student's head.

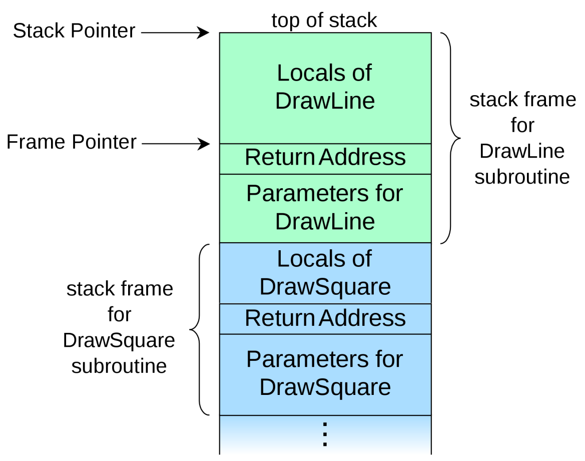


Figure 1: Call Stack (from Wikipedia). [Image credit: R. .S. Shaw, placed in public domain.]

Both Lewis [19] and Tunnell Wilson, et al. [28] ask students to hand-trace recursive programs using a substitution notional machine. Though they have somewhat different details, both use this to gain interesting insights into student performance. The latter work, in particular, also shows the shortcomings of hand-tracing, which motivates our tooling effort (see section 8).

Beyond the above literature, we also conducted an informal survey on stacks education through colleagues and social media. We find that:

- Many, including ones who use closures, do not use a clear notional machine for environments and stacks at all.
- Some use the conventional pictorial representation, which has many difficulties (section 4).
- Some use Python Tutor, which they acknowledge can be misleading or confusing for some of their programs.
- Some use an environment model [1]. This addresses issues of scope but not necessarily those of the stack and their interaction.
- Some use a substitution model [7], including its materialization in the DrRacket Stepper [3]. While this is technically correct and in principle covers all issues, they report that has numerous human-factors problems in both the notional machine and its visualization.

4 TECHNICAL BACKGROUND: (MIS)CONCEPTIONS

As a representative of a "typical" depiction of the stack, we use the image found on Wikipedia (fig. 1). Set against this type of representation, we discuss some of the issues that motivate this paper. We do this through four small, illustrative programs that help motivate why a student might need a notional machine, and for which an overly simple notional machine might be insufficient or even misleading.

Closures Outlive Stack Frames. Consider this program:

```
def f(x):
  return lambda y: x + y

p = f(3)
p(4)  # should produce 7
```

In the traditional stack drawing, `x` is allocated on the stack, but *disappears* once `f` finishes evaluating. Therefore, when `p(4)` evaluates, it is *reasonable* for a student to assume that `x` has been lost, and that the program results in an error. In fact, *we saw some students provide exactly this explanation* (see section 7). Thus, the traditional drawing is actively misleading.

Observe that the returned `lambda` is itself a first-class value whose lifetime exceeds that of the stack frame that created it. Therefore, the environment of that closure also needs to have an existence independent of that stack frame. Either `x` is initially allocated on the stack and subsequently moved to some environment structure, or it is allocated onto the heap right away.[3]

Control and Name Lookup Chain Differently. Now consider this program:

```
def f(x):
  return g(x + 1)

def g(y):
  return y + x

f(3)  # error: `x` is not defined
```

In this program, `x` is not in the scope of `g`, resulting in an error. However, in a stack drawing, `x` is present in `f`'s frame, which is the next frame. Furthermore, students have been conditioned that the stack "goes to the next frame after the current one", without necessarily realizing that this applies only to the control portion of the stack, *not* the data. Hence, it is easy to incorrectly conclude that this program also produces 7. Again, *we saw some students provide exactly this explanation*, sometimes justifying it using the stack drawing (see section 7).

Unfortunately, then, this representation leads students naturally to *dynamic* scope (when the value of a variable is given by the most recent binding in execution order, rather than from where it was bound in the source program). Dynamic scope has an infamous history in programming language design, having been present in the original LISP before it was removed. It was also present in original versions of Python and JavaScript before having to be steadily excised from them. In JavaScript, especially, its presence is a source of security attacks. Therefore, it is important for students who might *design* languages (which is a premise of the particular courses being studied studied in this paper) to be cognizant of and avoid making this mistake.

At a mechanical level, the problem here is a clash between the two purposes of the stack: to represent control and to represent the environment. These two entities *chain differently*. Control chains in the order of frames in the stack. The environment, however, chains to the environment that was present at its point of creation. The traditional stack drawing, and its interpretation, largely or

[3]This distinction corresponds to what different language implementations also do [2]. However, a notional machine is free to use a model that does not correspond to the underlying implementation.

entirely neglect the latter, inducing or reinforcing this dangerous misconception.

Depicting Fine-Grained Return is Tricky. Third, consider this program:

```
def lucas(n):
  if n == 0:
    return 2
  elif n == 1:
    return 1
  else:
    return lucas(n - 1) + lucas(n - 2)

lucas(5)
```

In principle, each stack frame has a return *address*. However, this is difficult to present in a useful, human-comprehensible manner, so many visualizations instead point to a *line of code*. Unfortunately, where there are multiple possible calls on a line, it can be very difficult to keep track of which calls are in progress and which are pending, and what the value is of the computation so far. Such visualizations are therefore not very useful even for classic program comprehension tasks like tracing recursion.

Modern Control Requires Multiple Stacks. Finally, consider this program, which uses a Python generator:

```
def nats():
  n = 0
  while True:
    yield n
    n = n + 1

ns = nats()
next(ns)
next(ns)
next(ns)
```

How is a student to make sense of this computation? At the point where the generator `yield`s, the generator does not stop computing the way a function would; rather, it suspends its computation. Thus, a student needs to accurately track the current state of the generator *separately* from that of the rest of the program. (Observe that in Python Tutor, the value of `n` *disappears* from the heap between calls to `next`, which is inaccurate and hence misleading.)

Moreover, generators in Python are quite simple. In some languages, generators can call other functions, which can perform the `yield`ing. Therefore, each generator needs *its own local stack* to keep track of what remains to be done.

Of course, even in Python, other operations require multiple stacks, e.g., threading. However, Python Tutor does not support threading (the "unsupported features" document includes "Multi-threaded / concurrent / asynchronous code"). Therefore, it does not need to confront this issue. However, students who are going to work with control features used in modern languages need a way to depict and hence understand them.

Summary. These programs demonstrate that even quite simple examples, of the sort that can occur early in some tertiary education curricula, demand quite a rich notional machine. In the absence of one, students either have no ability to explain program behavior or, worse, arrive at the wrong conclusions. Unfortunately, the traditional stack presentation is inadequate or even misleading for all these examples. (The last class of issues can be addressed using multiple stacks, but each of those stacks is subject to the earlier problems.)

PAPER PLAN

In section 5, we describe the two contexts in which we conducted our studies. Next, we describe the overall structure of our study: section 6. We then describe the results of the pre-test (section 7) and post-test (section 9). In between (section 8), we describe the tooling that we introduced with the hope of improving the quality of post-test responses. After presenting threats to validity (section 10), we discuss the research questions and other issues in section 12.

5 STUDY CONTEXT

We ran studies at two universities in the USA. At both, the studies were conducted in the context of an upper-level course on programming languages in which students wrote interpreters that implemented the features studied here. These do not teach *programming* but rather assume knowledge of it, and instead study the design and implementation of *languages*, following ideas from section 2. We describe them in turn.

Priv. is a highly-selective private university. It runs on a semester schedule; the instruction time runs for three months. Some entering students have programmed only in mainstream languages (primarily Java and some Python), while others (a majority) have also programmed in Racket, OCaml, Java, Pyret, and other functional languages. About 75% students have also taken one or more second-year courses that use C and explore low-level computer representations (including the details of the stack). The post-graduate students (especially PhD students) come from other institutions and therefore may have a more varied background.

In Fall 2021,116 students completed the course. Students enter the class with a variety of backgrounds. 72% had finished two or three years in college (with extensive programming background); about 11% were post-graduate or other students. Only 16% were second-year students, but most of these had extensive high school computing. Many students had also done one or more summer internships in industry.

Instructionally, the course used both the interpreter and ml approaches described in section 2. It also used other methods (such as quizzes) to try to develop students' mental models of programming language semantics. A particularly constant theme in the class was the importance of avoiding dynamic scope. This was reinforced through interpreters, mystery languages, as well as other activities, and through extensive lecturing (to the point where, when teaching assistants prepare recruiting materials, they use the tagline, "Want to join the crusade against dynamic scope?"). The course also made use of the tooling presented in section 8, using it live in several lectures as well as in lecture notes.

Since students are free to drop the course even late in the semester, we limited our analysis to students who had completed both the pre- and post-tests. We also made the post-test optional, to reduce the workload on students near the end of the semester. Thus, in the end we had 69 student responses to evaluate. We do not believe this materially affects the result of this *formative* study. The issues we find are a clear *lower bound* on the set of problems that students might have in the class.

Pub. is a public university. It runs on a quarter schedule; the instruction time runs for 10 weeks. The course is required. It is indicated for students at the end of their second year of study, but is taken later by many. Indeed, as it is not a prerequisite for any other courses, students often take the course in their last quarter.

Students entering this course have done required work in Python, Java, C, and Assembly, with several courses that discuss stacks at both the C and Assembly level. Typically, about a tenth of the students have some prior self-guided exposure to functional programming. In Winter 2022, running from January through March 2022, the studied section of the course held 27 students, all undergraduates; one had sophomore standing, three had junior standing, and the remainder all had senior standing.

Pub's curriculum in this course is a much purer representative of the interpreter approach, following its textbook [17] closely and focusing almost entirely on the process of understanding language features by implementing them as part of an interpreter. The course did not use mls or any stack-modeling tool.

Sampling. Due to the large amount of effort it takes to assess each student submission on each problem, for this formative study, we randomly sampled 50% of student submissions at each of the institutions from each of pre- and post-tests. However, we did also eyeball the submissions that were not assessed deeply to confirm that there were no surprising results that would materially throw off our findings.

6 STUDY DESIGN

Our study had two rounds. Though we use the term "test" below, students were not given any course grade in return for this work. At Priv, the instruments and rounds were modified on-the-fly based on prior student responses. The instructor, however, has a principle of announcing all graded work before the semester begins and adhering to these dates. Therefore, this work could not be administered for a grade. Pub offers three sections of the course that have to agree on a shared grading schema, and we had access to only one section. At both institutions, in return for the lack of grading, we expect vastly less potential for academic dishonesty, giving us a much purer view of student performance. Furthermore, from reviewing student responses, we can see that students took the work seriously; even though their work contained mistakes, it was not frivolous.

Pre-Test. The pre-test was administered early in the course, before students were exposed to any course material on stacks or interpreters. This was to obtain a sense of their incoming notions and conceptions of this material. Concretely, students were given the programs shown in fig. 2 and asked to draw the stack (through any means they wish) at the point(s) when the programs paused.

How do we unambiguously indicate the point of a pause? We were concerned that various graphical notations may be unclear. For that reason, we included the following pause procedure with every program:

```
fun pause():
  # suspend here before continuing computation ...
  0
```

This has no effect on the computations (because it returns `0` to positions that perform addition). We believe this gives us a clear and consistent way to describe a moment in the computation. This appears to have worked, and we did not notice any confusion caused by this choice of notation.

The programs were designed to capture the following:

(1) Simple function call behavior.
(2) Handling of global and local variables.
(3) Keeping track of the state of a loop.
(4) Dynamic scope (students should consider this program erroneous).
(5) Function calls in a parameter position.
(6) Recursion. We intentionally chose a program that was not a standard recursive function they might have seen before, so they could not depend on recall.
(7) Nested functions.
(8) Functions returned as values, requiring closure creation.

Students were explicitly told that the programs were in a "hypothetical infix syntax meant to represent an idealized language reminiscent of Python, Java, OCaml, Scheme, etc.". This was intentional: we wanted them to use a standard model of evaluation and not get bogged down in the whimsical details of specific languages such as Python [22]. Thus, the instructions also said, "In case you happen to know about some peculiar behavior of any of those languages, don't assume it here, because this is meant to represent what is *common* to all these languages" (emphasis in original). Indeed, we did not get any inquiries from students about the meanings of the syntax they were given, and we observed only a few instances where the syntax might have been a cause of confusion. (Concretely, though the instructions explained it, some students were not accustomed to seeing a "functional" style where the body of a function automatically returns a value; they expected to see an explicit `return` statement. But most students were able to infer this without difficulty.)

Students were asked to do three things for each program: "draw the stack" at the point where the program calls `pause`, indicate the final value produced by the program, and describe whether the stack helps them in determining the answer (and if so, how).

Our interest is in the conceptions students have of the stack coming in. Students who have either never studied it formally (like some at Priv) or are do not feel familiar with it (at either institution) may well have various misconceptions, but those are not relevant to our study. We wanted to focus on those students who self-identified as being familiar with stacks. Therefore, the study first asked them whether they were "familiar with the computer's run-time stack, as covered in systems courses". To avoid confusion, it added, "Note: not the generic stack *data structure* from introductory computing". Students who answered negatively were led to the egress. Those who answered affirmatively were asked to confirm both parts again. Our findings are only from those students who self-identified as being familiar with stacks.

For each of the programs in the test, students answered three questions: What the stack looks like at the point (or points!) where

```


PROGRAM 1
---------

fun f(x,y):
  x + g(y + 1)

fun g(z):
  z + pause() + z

f(2, 4)

---------
PROGRAM 2
---------

z = 19

fun f(x,y):
  q = 9
  x + pause() + y + q

f(2,4)

---------
PROGRAM 3
---------

fun f(x):
  ans = 0
  for y in range(0, 5):  # y has values 0 through 4
    if (y == 2) or (y == 3):
      ans = ans + g(y + x)

fun g(z):
  pause() + z

f(1)

---------
PROGRAM 4
---------

fun f(x):
  g(3)

fun g(y):
  pause() + y + x

f(2)
```

```


PROGRAM 5
---------

fun f(x):
  g(g(x - 1) - 1) - 2

fun g(y):
  pause() + 1 + y

f(9)

---------
PROGRAM 6
---------

fun f(x):
  if (x < 0):
    pause() + 4 + x
  else:
    f(x - 2)
  end

f(3)

---------
PROGRAM 7
---------

fun f(x):
  fun g(y):
    pause() + x + y
  g(99)

f(4)

---------
PROGRAM 8
---------

fun f(x):
  fun g(y):
    pause() + x + y
  g

fun h(i,x):
  i(4)

h(f(3),5)
```

Figure 2: Pre-Test Programs

pause is called, what the program produces (which could include an error), and whether the stack diagram helped them figure out what it produces.

Post-Test. The post-test was administered after students had completed all the educational components. The programs used in the post-test are shown in fig. 3. These were based on what issues were left from earlier rounds at Priv, and capture these important aspects:

(1) Multiple function calls, with the return of at least one function before the use of pause.
(2) Function calls in a parameter position.
(3) Simulation of i/o and global state mutation, showing the suspension and resumption of computation.

(As such, there may be—and indeed, appear to be (see section 9)—more issues left at Pub.)

At Priv, the post-test was given in the third month of the course, near the end of the semester. This was actually the *fourth* round of the study. Two intermediate rounds occurred in preceding months. Both used simpler forms of the stack visualization tool described in section 8, and students were sent auto-graded feedback and shown reference solutions. Thus, students taking the post-test had received feedback about prior mistakes and had had a chance to practice with the tool. Of course, there is no reason to believe all students paid particularly close attention to the feedback, especially because it was not tied to any course grade. To keep the workload reasonable we consolidated and eliminated questions based on the feedback from each round, which is why the post-test instrument is much smaller; including more questions may very well expose more misconceptions.

At Pub, the post-test was given in the final week of the course. To harmonize with the other sections, the instructor had *not* used the stack visualization tool in the course. However, the Pub students, like those at Priv, had implemented interpreters that made explicit use of environments, including a careful examination of the role of closures, the necessity of capturing the environment at the time of closure creation, and the importance of supplying that captured environment to the evaluation of the called closure's body. This had been the explicit subject of lectures as well.

Evaluation. For both rounds, student work was evaluated by both authors, who used the methods of grounded theory to create rubrics. The pre-test rubric is shown in fig. 9 and fig. 10. The evaluation elements were grouped into five "axes". After 3 rounds over the course of a month, we achieved Cohen κ scores of 0.84, 0.90, 1.0, 1.0, and 1.0 along the five axes. These axes are indicated by textual prefixes: e.g., the b/ in b/missing indicates the "behavior" axis.

Not all codes can apply to all programs. Whenever a rubric element contains numbers in parentheses (such as b/pop(358)), it means the code should only occur in the programs listed (in this case, 3, 5, and 8).

Both authors then used this rubric as a starting point for the post-test evaluation. We examined student submission samples to look for codes that should be either added, dropped, or modified. The end result was the rubric shown in fig. 11 and fig. 12, including an additional axis to capture whether students elected to use the Snap*!*-based tool in their post-test. Developing this rubric required 3 rounds over the course of 3 weeks, and we ended after achieving κ scores of 0.74, 0.76, and four axes with perfect agreement.

When a code is listed as unchanged, it means we retained the same meaning and even the same prose, except for modification to reflect the new problem numbers.

What changed in the latter rubric? Because of the semester of educational intervention, one in which—in particular—students implemented interpreters with explicit environments and closures, we expected students to do a much more detailed job of environments; they did, resulting in codes for environments (e/) (fig. 11). At Priv, issues with closures seemed to disappear, so we dropped those codes. (This was less true of Pub, but we used the same instrument and rubric for both.) The other big change is that, having given Priv students a representation of the control context, we expected students to use it. This led to significant extension of the c/ codes (fig. 12) to cover their use and misuse.

7 FINDINGS: PRE-TEST

At Priv, 86 of 119 students indicated they were familiar with the stack. All but two reported having had one or both of the intermediate-level systems courses at Priv or a comparable course elsewhere, and in some cases had also had upper-level systems courses (like compilers) that had discussed the stack.

At Pub, 25 of 27 students indicated that they were familiar with the stack. Every one of the students reported having taken the same preparatory course, an introduction to C and basic UNIX system calls.

Students generally got the answers to the given programs right, with these notable exceptions:

(1) Some students were thrown off by the dynamic scope problem, for which they incorrectly gave the wrong answer, in some case directly attributing it to the stack diagram (e.g., "the variable x was in scope, since it was in a higher/parent stackframe"). Even more interesting were some of the written answers from students who got the *right* answer. Figure 4 gives some representative student comments and our observations on them.
(2) On the last problem, over a third of the respondents at Priv and more than a half at Pub thought it produces an error. We have seen this in other settings also, where students assume a function name not followed by a call must be an error, rather than the function being referred to as a value.
Of course, we can simply put this down to lack of experience with functional programming. Much more interesting is that even the *correct* answers largely had little to no idea about how the stack applies to such a program. As comments in fig. 4 and their ilk indicate, students were only used to thinking of the stack for "stack-based" languages like C. Program 8 requires the creation of a closure, which effectively "extracts" the binding part of the stack frame onto the heap. Students were clearly not trained to handle such programs.

Quantitatively, fig. 5 shows the results from coding. We present numbers as percentages because some codes apply to only a few problems (e.g., b/missing only applies to programs 3 and 5), so

```
PROGRAM 1                               PROGRAM 2
---------                               ---------
fun k(i, j):                            fun f(x):
  i + g(j - 1) + h(j)                     g(k(x))

fun g(w):                               fun k(w):
  w + 7                                   (w + pause()) - 4

fun h(a):                               fun g(z):
  z = a + 1                               z + 1
  a + pause() + z
                                        f(2)
k(8, 14)



---------
PROGRAM 3
---------
prompt = ""
response = 0

# Observe that reading a value from a user requires pausing the program.
# To avoid getting into all the details of input-output, we simulate it as follows:
# The prompt string provided is assigned to `prompt`.
# Assume this is sent to the user, who types a response.
# Their response is assigned to `response`.
# The `read` procedure then returns this value, as if the user had typed it in.

fun read(new_prompt):
  prompt := new_prompt
  pause()    # Assume that `response` is set BEFORE the call to `pause` begins
  response

print(read("First number") + read("Second number"))

During the evaluation of this program, assume that during the first call to pause
the global variable response is set to 11, and during the second call it's set to 4.
```

Figure 3: Post-Test Programs

absolute numbers would look artificially small.[4] We round to the nearest whole number for simplicity.

Some data stand out. Most of all, we see numerous instances of problematic codes like B/PILEUP, BND/NOPARAMS, and BND/NOLOCALS. Many students did not consider the control portion of the stack at all (C/NONE). The fact that about 20% of PRIV students drew contexts is probably explained by the fact that one of the introductory courses teaches functional programming using a substitution-based notional machine [7]. In the other direction, all PUB students were required to have taken a systems class, which probably explains the much larger ratio choosing byte representations.

[4]There is a slight imprecision here: a handful of codes applied to other problems, e.g., if a student asserted that a program doesn't run at all and hence there is no stack, this too would be a B/MISSING. However, there were only a few of these cases, so the numbers are still very broadly representative.

Also, in an open coding exercise such as this one, there are bound to be a number of uncategorizable errors, marked here with B/OTHERERR. In a number of these, certain frames of the stack are simply omitted without obvious cause. In quite a number of them, students imagined that a program containing the code `f(g(x))` would have a stack showing a call to `g` inside a call to `f`. At least one student indicated that the stack would be nonexistent because the program could be compiled to run entirely in registers. Finally, several of them produced representations that we were unable to map to any notion of a "stack" whatsoever.

8 STACK REPRESENTATION TOOLING

Pre-Test Lessons. One highlight of the pre-test responses is the utter lack of standardization in stack drawings we saw. Even though prior classes had used fixed notations, essentially no two students

Student Comment	**Commentary**
"yes? it made me think `x` was still available to `fun g()`, though I am not sure how the scope is defined in this assignment"	A few students found the traditional stack diagram actually inspired a dynamic-scope misconception.
"Not really, since scoping rules are the determining factor here and a stack by itself seems to imply global scope"	This is similar to the previous comment: the stack seems to suggest all variables are in scope, even though the student realizes they *should* not be.
"It helped to see `f` and `g` as two different stack frames to see that `x` was not in scope for g."	Several students wrote answers that suggest that a function can never refer to variables from another frame, which is false in the presence of first-class functions.
"The stack actually created some confusion for me on this one as I realized I wasn't sure how to show what part of the stack a function had access to. I believe this would error, but just by looking at the stack (at least how I have been drawing them), that is not immediately apparent."	Here the student realizes the stack *ought* to help them find the scope problem, but can't see how.
"the way I drew it, it wasn't useful for determining scope. i'm not particularly sure how the stack and scope interact."	Some students found themselves simply confused about scope by the stack diagram, even though the stack ought to be useful precisely for determining what is and isn't in scope.

Figure 4: Student Comments from the Pre-Test

Code	Priv	Pub
s/bytes	50%	82%
s/logical	46%	9%
b/missing	13%	36%
b/extra	1%	1%
b/pop	68%	30%
b/pileup	8%	15%
b/tailcall	3%	3%
b/othererr	7%	30%
c/addr	33%	36%
c/ctxt	20%	0%
c/none	46%	55%
bnd/params	78%	77%
bnd/noparams	20%	14%
bnd/locals	88%	73%
bnd/nolocals	18%	23%
bnd/noclosures	1%	1%
clo/opaque	47%	27%
clo/clear	18%	9%

Figure 5: Qualitative Analysis of Pre-Test Responses

drew stacks the same way. Many seemed to invent notations on the fly, use the stack as a scratchpad, and so on. Out of respect for the students we choose to not reproduce any of their images here, but some are quite extreme and took a great deal of time for the two (expert) authors to decipher; others are excessively simplistic.

Another highlight is how few students had means to represent what remains to be done in the computation. Many did not even seem to realize this is (half) the job of the stack! Even many of their written comments suggested they did not have a clear understanding of this point. Unfortunately, this means that they probably do not have a good model for control operations such as threads, coroutines, generators, etc.

Design Goals. We therefore devised a new visual representation. We focused on showing the instantaneous *state* of the system (stack, environment, and heap). A full notional machine of course also requires rules of program execution, but our students already had significant prior programming experience. For a more elementary class, our representation can easily be combined with a model of execution.

We wanted students to be able to describe system states for us. This meant we had to minimize tedium, error, and boredom, to avoid problems noticed in prior work [28] where students got bored and skipped steps.

Our ultimate goal was therefore to create a tool that would (a) facilitate state rendering, (b) provide a standard notation, and (c) nudge students towards representing what remains to be done in the computation; our hope was to also direct students away from misconceptions.

This implied several features:

(1) We wanted to keep a familiar "stack" feel, with the ability to "push" and "pop" frames.
(2) We wanted the stack frames to refer to familiar components such as the parameters and local bindings. We also wanted each frame to have an explicit record of what remains to be done.
(3) We wanted to avoid the dynamic scope misconception (section 4). This required making the variable resolution process explicit, and separate from the stack order.
(4) We wanted to make it possible to capture how closures keep track of variables bound in their lexical context. This is difficult to do if all variables are allocated on the stack, because when a call finishes, those variables disappear. Instead, these bindings must reside outside the stack frame.

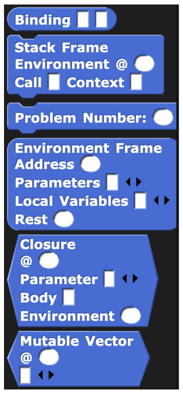


Figure 6: Palette of Notional Machine State Blocks

Tooling Choice. We were initially inspired by the notion of Parsons problems [21], and intended to repurpose a Parsons library—using linear stack frames rather than program statements. Because we wanted students to actively fill in parts of the frames and even whole frames, we considered the faded Parsons problems toolkit [29]. However, this is still a fairly restrictive environment, without room to accurately depict detached environments as well as closures and other heap-allocated values.

Instead, we successfully repurposed Snap*!* [15] for our purpose. Snap*!* blocks have all the features we want—easy to construct, easy to manipulate, unambiguous, visually distinctive, available in a structured format for automated assessment, etc.—while being flexible and not limited to just one dimension.

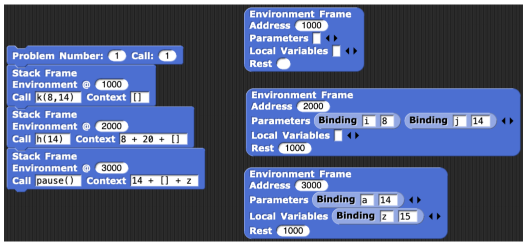
Block Design. We created a Snap*!* configuration that gets rid of all the pre-existing blocks and has only a small number of custom blocks that we defined, shown in fig. 6:

Stack Frames These intentionally do not record variables locally, but rather in a separate Environment Frame, identified by an address. This enables bindings to outlast a function call, as required by closures. The Call field records the function call that caused the current frame to appear on the stack, and the Context reflects what remains to be done in the computation (explained in more detail below). These are chosen to be blocks that can be "stacked" vertically. The Environment reference must be numbers.

Environment Frames These blocks record the parameters and local variables. Their Rest field is used to chain references; this chaining can follow lexical scope rather than the calling order (to avoid dynamic scope). These blocks are intentionally chosen to not be "stackable". Both Address and Rest are constrained to be numbers.

Closures These are heap-allocated values that contain the code of a function and "close over" its defining (lexical) environment. The address and Environment fields are similarly required to be numeric. All Parameters and Local Variables slots are arrays of Bindings. Every Call and Body slot is code.

Mutable Vectors These represent mutable, heap-allocated data. They were used in class but not in this paper.

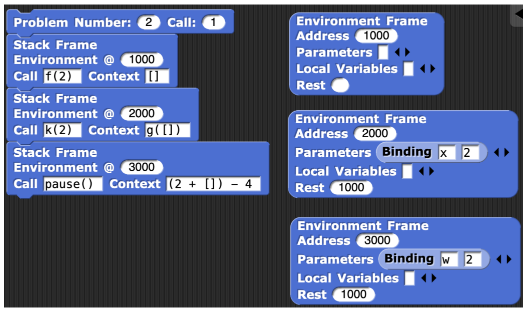

Problem Number: 1 Call: 1
Stack Frame
Environment @ 1000
Call `k(8,14)` Context `[]`
Stack Frame
Environment @ 2000
Call `h(14)` Context `8 + 20 + []`
Stack Frame
Environment @ 3000
Call `pause()` Context `14 + [] + z`

Environment Frame
Address 1000
Parameters
Local Variables
Rest

Environment Frame
Address 2000
Parameters Binding `i` `8` Binding `j` `14`
Local Variables
Rest 1000

Environment Frame
Address 3000
Parameters Binding `a` `14`
Local Variables Binding `z` `15`
Rest 1000

Solution to Problem 1

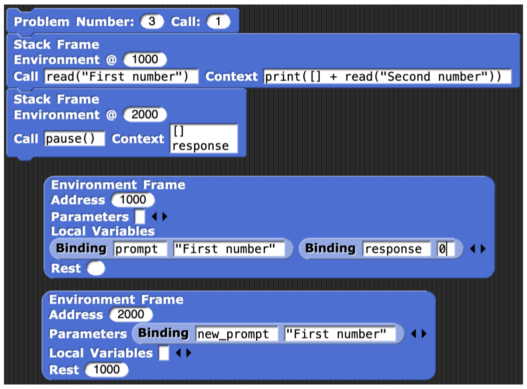

Problem Number: 2 Call: 1
Stack Frame
Environment @ 1000
Call `f(2)` Context `[]`
Stack Frame
Environment @ 2000
Call `k(2)` Context `g([])`
Stack Frame
Environment @ 3000
Call `pause()` Context `(2 + []) - 4`

Environment Frame
Address 1000
Parameters
Local Variables
Rest

Environment Frame
Address 2000
Parameters Binding `x` `2`
Local Variables
Rest 1000

Environment Frame
Address 3000
Parameters Binding `w` `2`
Local Variables
Rest 1000

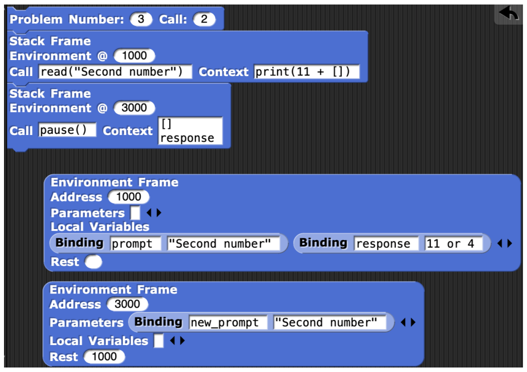
Solution to Problem 2


Problem Number: 3 Call: 1
Stack Frame
Environment @ 1000
Call `read("First number")` Context `print([] + read("Second number"))`
Stack Frame
Environment @ 2000
Call `pause()` Context `[]`
`response`

Environment Frame
Address 1000
Parameters
Local Variables
Binding `prompt` `"First number"` Binding `response` `0`
Rest

Environment Frame
Address 2000
Parameters Binding `new_prompt` `"First number"`
Local Variables
Rest 1000

Solution to Problem 3, First Stack


Problem Number: 3 Call: 2
Stack Frame
Environment @ 1000
Call `read("Second number")` Context `print(11 + [])`
Stack Frame
Environment @ 3000
Call `pause()` Context `[]`
`response`

Environment Frame
Address 1000
Parameters
Local Variables
Binding `prompt` `"Second number"` Binding `response` `11 or 4`
Rest

Environment Frame
Address 3000
Parameters Binding `new_prompt` `"Second number"`
Local Variables
Rest 1000

Solution to Problem 3, Second Stack

Figure 7: Solutions to Post-Problems

Problem Number These blocks are to help the auto-grader.

These blocks were introduced incrementally at Priv. The first round (pre-test) had no blocks at all. In the second round students were given Stack Frames with Contexts but with the content of the Environment Frame inlined, much like the standard stack diagram. In the third round the Environment Frame separated from the stack, and the Closure data structure was introduced. Mutable Vectors were introduced for use during lecture before the fourth round (post-test).

Based on these blocks, fig. 7 shows the solutions for the post-test problems. It is important to remember that these Snap*!* configurations are *not* programs; it does not make any sense to "run" these. They are just visual depictions of instantaneous program states; i.e., we are using Snap*!* purely as a drawing medium.

We chose numbers to represent references for two reasons. First, very simply, even if we had preferred drawing direct arrows, this is currently not possible with Snap*!*. Second, numeric references do confer a degree of authenticity, since they reflect how memory addressing works.

Utility for Instruction and Assessment. We note that this tool is very useful in both in-class instruction and in written notes. Due to draggability and the small amount of typing, it was very useful for generating stacks on-the-fly in class. It also became easy to generate (and, by saving the XML file, modify) stacks for inclusion in notes. In addition, it may help students to use *exactly* the same imagery in the notes, on screen in class, and in the student assignments. (The lack of tooling may explain the wide variation we saw in the pre-test. This leaves students to their own devices, resulting potentially in both lack of reinforcement and divergence in presentation.)

This tool also greatly simplifies providing feedback. It is essentially a drawing medium that produces unambiguous, structured output. Thus, students can quickly get automated responses. It also aids manual assessment by producing standard and clear visuals.

Contexts. Finally, we explain the Context field of a Stack Frame. Evaluation contexts are a standard mechanism in programming language semantics [9] for expressing what remains to be done in the computation, i.e., both the binding and control part of the stack. We adapt this idea to our context. At every function call, the Context is an expression that shows what remains to be done *in this function* and Call shows what the newer frame will compute. This avoids the "return line" problem (section 4).

The expression `[]`—called the "hole"—stands for where evaluation is currently occurring, i.e., the hole will be filled in by whatever is returned by the newer stack frame. All sub-expressions that have finished evaluating are replaced with their values; all those waiting to evaluate are left as source (e.g., `14 + [] + z` in Problem 1). For instance, when evaluating `lucas(4)` (section 4), we will see the context `[] + lucas(n - 2)` with the Call `lucas(3)`, and later `4 + []` (where `lucas(3)` is 4) with the Call `lucas(2)`. In particular, in the presence of mutable variables, it is important to *not* substitute variables with values prematurely, since those values may change before the variable is eventually evaluated.

A full discussion and formalization of these contexts can be found in other work [9] and is outside the scope (and available space!) of this paper. In addition to the explanations above, the reader can also see examples of their use in fig. 7. In particular, we note that integrating this into the course at Priv was effective and the notation was easily within reach of students, as section 9 shows.

Code	Priv	Pub
scorable	95%	83%
snap	82%	0%
s/bytes	0%	42%
s/logical	95%	42%
b/missing	0%	8%
b/extra	1%	0%
b/pop	79%	67%
b/pileup	12%	13%
e/separate	85%	0%
e/chain	55%	0%
e/badchain	49%	83%
c/addr	0%	0%
c/ctxt	95%	15%
c/nohole	7%	4%
c/subright	1%	0%
c/other	12%	0%
c/none	0%	65%
bnd/params	94%	63%
bnd/badparams	1%	21%
bnd/locals	97%	83%
bnd/badlocals	3%	0%
bnd/toplevels	85%	46%
bnd/badtoplevels	10%	33%

Figure 8: Qualitative Analysis of Post-Test Responses

9 FINDINGS: POST-TEST

Students were asked to present the outputs of programs. In contrast to the pre-test, students essentially predicted the answers correctly (excepting some arithmetic mistakes).

We therefore focus on coding the post-test state presentations. These are shown in fig. 8. As before, codes are shown as a percentage of students who had them, and normalized for the number of problems that could have manifested that code. Thus, with 83% of Pub solutions scorable, the 83% with a c/badchain tells us that *all* scorable solutions had a bad environment lookup chain!

Overall the results are fairly good for Priv, which is perhaps unsurprising (and a relief) given multiple rounds of practice (including feedback) with increasingly richer notional machines. We see high use of Snap*!*, and quality use of contexts. Students remain somewhat confused about where to put top-level variables, which we can attribute to our pedagogy, which may have left this unclear. Two issues stand out. One is the number of students who *still* had b/pileup (i.e., treating the stack like a journal rather than as a structure with pops in addition to pushes). The other is the large number of b/badchain. Because correct environment chaining in the interpreter was an explicit topic of class lecture, textbook coverage, as well as implementation, and students were expected to write a number of tests that covered this as well, we see a clear failure to transfer from the various interpreter-related settings—as well as the various exhortations against dynamic scope—to this setting.

The results are much worse for PUB. Because PUB did not have several intermediate rounds, didn't use Snap*!* in class, and didn't use all the educational techniques that PRIV did, it is not surprising to see a difference. That said, PUB is a much more accurate evaluation of a "pure" interpreter-based class (see section 2). These numbers show that such an approach leaves a great deal wanting if we want students to understand the features discussed in this paper.

Identifying which of the interventions at PRIV were necessary to close this gap, and also what PRIV needs to do to fix its remaining issues, remain open tasks for future work.

10 THREATS TO VALIDITY

This work naturally has many threats to validity. We discuss them below. In general, our work does not carefully tease out the effect of specific curricular interventions.

Internal Validity. Our studies were conducted without grade pressure. However, our reading indicated that students took the work seriously. (Student submissions were not anonymous.) By making the post-test optional at PRIV, we obtained responses from only a subset of students; expanding the population could affect the percentage of students exhibiting difficulty in either direction. Also, students might be learning about stacks from other concurrent courses; this could aid or hurt their performance on our instruments.

External Validity. Our work is clearly difficult to extract from our specific contexts. The nature of the institutions and their students, the students' programming background, and the particular intermediate systems programming courses they took, would surely have a big impact. Despite these differences, however, we find problems across the two institutions. This suggests the problems are unlikely to be isolated, and would benefit from a large-scale, multi-national, multi-institutional [10] study.

Ecological Validity. Our use of small and artificial programs helps us keep the study manageable. It is tempting to expect that larger programs would only exacerbate these problems. However, the use of an artificial language not only reduces familiarity, it also robs students of tools (such as debuggers) they can use in practice. In general, it remains unproven that students actually *need* notional machines to understand and debug programs. Nevertheless, we find it difficult to imagine that misunderstandings of basic concepts like the stack would not translate into real-world difficulty.

11 RESEARCH ETHICS

Per Brown University's Institutional Review Board guidelines, our work does not require review. Nevertheless, we have applied standard research protections to protect our students.

12 DISCUSSION

Answering the Research Questions. Based on the above, we find the following answers to our initial research questions:

(1) At entry, we find that students have a somewhat poor understanding of stacks. They have difficulty with its relationship to scope, which is exhibited when asking them to reason about a program that relies on dynamic scope to run. Only 68% of the submissions from PRIV students and a mere 30% of submissions from PUB students correctly illustrate the "pop" stack operation that should occur when a function call returns. They are unable to relate programming concepts like closures to the stack. Finally, they largely overlook the control portion of the stack, which suggests that they might have trouble with advanced control features.
(2) After a term of instruction with just interpreters at PUB, student performance was distressingly similar to their pretest results, albeit with a pronounced swing toward frame-based (S/LOGICAL) representations. In contrast, at PRIV, where interpreters were augmented with MLS and direct instruction on the stack and use of our tool in class, students did much better. (Of course, there are likely also differences in the student population.) However, even at PRIV, students were not able to transfer knowledge about environments from interpreters (where they had implemented them explicitly). This forces a reconsideration of the utility and impact of interpreters, and suggests the need for more and different educational interventions.
(3) Our tooling was useful up to a point. At PRIV, students were exposed to it through multiple intermediate tasks (with the tool growing in complexity). Left to their own devices, 82% of them voluntarily chose to use it in their post-test. At PUB, where students were not given this instruction or experience, nobody used the tool. This suggests that students find it useful and will adopt it with practice, but will not do so otherwise.

Overall, we find our results very sobering. Despite intermediate coursework on systems programming that explains stacks in some detail, students have only a fragile understanding of them. In particular, their knowledge does not generalize even to concepts like closures.

Our study did not ask students to explain the behavior of more modern control constructs like threads and coroutines; based on what we have seen, we conjecture that doing so would have exposed even more difficulties. For the same reason, though we did not explicitly check for it, we strongly conjecture that students would struggle even to explain nested scopes, an idea that has been documented since at least 1964 [25, section 2.2.1.1.3].

Depicting References. In section 8, we discuss the use of numbers to show reference between components. We note there that these references could be drawn using arrows instead. This is not currently possible using Snap*!*. Between the benefits of Snap*!* and the argument for authenticity, we chose to not implement such a system for this study. However, doing so and comparing these representations would be very interesting. In particular, one can easily imagine students starting out with arrows and, after they have mastered the ideas, switching to numeric addresses to better understand the underlying machine representation.

Spatial and Numeric Factors. As fig. 7 shows, our official solutions tended to present Environment Frames in a neat, linear order, and with numbers evenly spaced apart. In retrospect, both seem unwise, because they seem to "stack up" the frames—precisely the phenomenon that led to dynamic scope misconceptions. Though a combination of interventions removed that at Priv, in their absence, this misconception might remain. We believe it would be healthier to place the Environment Frames in a more haphazard manner, and to use more "random" addresses.

Consequences for Interpreters. Interpreters are a valuable implementation technique and are even used in practice. Thus, they are a useful learning objective in themselves. Interpreters are also a constructionist (and hence constructivist) way of learning about programming languages. Students build the essence of languages, get to play with them, and can see how small changes in these implementations can have an impact on the resulting language.

However, our paper suggests that interpreters may have limited impact on conceptual understanding, which requires transfer from the implementation. While our evidence is preliminary, it suggests that people who are using interpreters with the hope of such transfer need to carefully examine whether they are getting the benefit they imagine.

ACKNOWLEDGMENTS

We thank Jens Mönig and the rest of the Snap*!* team for their superbly flexible tool, and Jens in particular for answering our questions. We are grateful to Elijah Rivera for figuring out how to customize Snap*!* and helping us with the process. We thank Barb Erickson for introducing us to Nate Weinmann and Nate for his help experimenting with his tools. We thank our students for patiently tolerating our experiments with these. Kuang-Chen Lu helped by reading the paper and catching mistakes. Juha Sorva generously answered our questions about his and others' work, and Ben Shapiro helped us better understand the paper's theoretical basis. This work was partially supported by the US National Science Foundation.

REFERENCES

[1] Harold Abelson and Gerald Jay Sussman. 1985. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA.
[2] Andrew Appel. 1991. *Compiling with Continuations.* Cambridge University Press.
[3] John Clements, Matthew Flatt, and Matthias Felleisen. 2001. Modeling an algebraic stepper. In *European Symposium on Programming.*
[4] Amer Diwan, William M. Waite, Michele H. Jackson, and Jacob Dickerson. 2004. PL-Detective: A System for Teaching Programming Language Concepts. *Journal on Educational Resources in Computing* 4, 4 (Dec. 2004). https://doi.org/10.1145/1086339.1086340
[5] Benedict du Boulay, Tim O'Shea, and John Monk. 1999. The Black Box Inside the Glass Box. *International Journal of Human-Computer Studies* 51, 2 (1999), 265–277.
[6] Rodrigo Duran, Juha Sorva, and Otto Seppälä. 2021. Rules of Program Behavior. *ACM Transactions on Computing Education* 21, 4 (Nov. 2021). https://doi.org/10.1145/3469128
[7] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs.* MIT Press. http://www.htdp.org/
[8] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. In *Communications of the ACM.*
[9] Matthias Felleisen and Robert Hieb. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science* 102 (1992), 235–271.
[10] Sally Fincher, Raymond Lister, Tony Clear, Anthony Robins, Josh Tenenberg, and Marian Petre. 2005. Multi-institutional, multi-national studies in CSEd Research: some design considerations and trade-offs. In *SIGCSE International Computing Education Research Conference.* https://doi.org/10.1145/1089786.1089797
[11] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1997. DrScheme: A Pedagogic Programming Environment for Scheme. In *International Symposium on Programming Languages: Implementations, Logics, and Programs (Springer Lecture Notes in Computer Science, 1292).* 369–388.
[12] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. 1992. *Essentials of Programming Languages.* MIT Press.
[13] Philip Guo. 2021. Ten Million Users and Ten Years Later: Python Tutor's Design Guidelines for Building Scalable and Sustainable Research Software in Academia. In *ACM Symposium on User Interface Software and Technology.* 1235–1251. https://doi.org/10.1145/3472749.3474819
[14] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education. In *ACM Technical Symposium on Computer Science Education.*
[15] Brian Harvey and Jens Mönig. 2010. Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists?. In *Constructionism.*
[16] Samuel Kamin. 1990. *Programming Languages: An Interpreter-Based Approach.* Addison-Wesley.
[17] Shriram Krishnamurthi. 2006. *Programming Languages: Application and Interpretation.*
[18] Shriram Krishnamurthi and Kathi Fisler. 2019. Programming Paradigms and Beyond. In *The Cambridge Handbook of Computing Education Research*, Sally Fincher and Anthony Robins (Eds.). Cambridge University Press.
[19] Colleen M. Lewis. 2014. Exploring variation in students' correct traces of linear recursion. In *SIGCSE International Computing Education Research Conference.* https://doi.org/10.1145/2632320.2632355
[20] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and their Computation by Machine. *Commun. ACM* 3, 3 (1960), 184–185.
[21] Dale Parsons and Patricia Haden. 2006. Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses. In *Australasian Conference on Computing Education.*
[22] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: The Full Monty: A Tested Semantics for the Python Programming Language. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications.*
[23] Josh Pollock, Jared Roesch, Doug Woos, and Zachary Tatlock. 2019. Theia: automatically generating correct program state visualizations. In *SPLASH Education Symposium.* https://doi.org/10.1145/3358711.3361625
[24] Justin Pombrio, Shriram Krishnamurthi, and Kathi Fisler. 2017. Teaching Programming Languages by Experimental and Adversarial Thinking. In *Summit on Advances in Programming Languages.*
[25] B. Randell and L. J.Russell. 1964. *ALGOL 60 Implementation.* Academic Press.
[26] Juha Sorva. 2012. *Visual Program Simulation in Introductory Programming Education.* Ph. D. Dissertation. Aalto University, Department of Computer Science and Engineering.
[27] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *ACM Transactions on Computing Education* 13, 2, Article 8 (July 2013), 31 pages. https://doi.org/10.1145/2483710.2483713
[28] Preston Tunnell Wilson, Kathi Fisler, and Shriram Krishnamurthi. 2018. Evaluating the Tracing of Recursion in the Substitution Notional Machine. In *ACM Technical Symposium on Computer Science Education.*
[29] Nathaniel Weinman, Armando Fox, and Marti A. Hearst. 2020. Exploring Challenging Variations of Parsons Problems. In *ACM Technical Symposium on Computer Science Education.* https://doi.org/10.1145/3328778.3372639

Notions of the stack:

s/bytes	The stack is a sequence of bytes or bits. Frames are not delineated, contexts (if present) are indicated by return pointers, environment values may not be labeled.
s/logical	The stack is a logical part of a step in a notional machine. Frames are clearly separated, contexts may be represented as source code with holes, environment values are definitely labeled. Regions of the stack may be labeled with function names.

Behavior of the stack:

b/missing(35)	This solution is missing at least one full stack. That is, the solution missed a call to `pause`.
b/extra	This solution contains an extra stack. That is, the solution imagines an extra call to `pause`.
b/pop(358)	Stack frames are created when a function is called, and deleted when the function returns. In problems where no function has returned when the pause is called, no popping can be observed, and no box should be checked. Absence of this code on problems 3, 5, or 8 probably implies that b/pileup should be checked.
b/pileup(358)	Stack frames are created when a function is called, and never deleted. These solutions suggest that frames are essentially a record of every function that is ever called. Absence of this code on problems 3, 5, or 8 probably implies that b/pop should be checked. Some solutions include separate slots for the loop variable, e.g., `y = 2` `y = 1` `y = 0` These solutions should be coded as b/pileup unless it's very clear from the presentation that the bindings are all contained in a single stack frame.
b/tailcall(468)	Stack frames are created when a function is called, but may be replaced by a tail call.
b/othererr	Some other error is present. Make a note of what it is.

Handling of contexts:

c/addr	Opaque address: the solution uses the words "return address" or similar language to indicate that the function returns to some earlier point of execution. Note that this is definitely distinct from the frame pointer, pointing to the earlier frame.
c/ctxt	Source context: the solution represents context using a short snippet of source code containing a hole or other placeholder. Ideally this would be the full context, but in the presence of multi-line functions, few or no students include the following lines. The most obvious example of this occurs in the loop example, program 3. Solutions that include the call made as part of the context should receive this mark, e.g. those with `pause() + 4`.
c/none	No context: the solution omits any reference to context, containing only bindings and/or frame pointers.

Figure 9: Rubric for Pre-Test, Part 1. Continues in fig. 10.

Handling of bindings, both parameters and local variables:	
BND/PARAMS	At least one parameter binding appears in the stack. This includes solutions that show only the names of the bindings, and not their values.
BND/NOPARAMS	At least one parameter value does not occur in the stack. This includes solutions that show the call as e.g. `f(2,4)` (that is, where the parameter values can be deduced from the drawing but are not shown as taking independent space in the stack). This also includes solutions where the student has performed substitution on a copied body of the called function.
BND/LOCALS(23)	The problem contains at least one local or top-level binding that appears in the solution. Note that global bindings such as `z = 9` should be regarded as local bindings at the outermost scope. Note further that function bindings (including top-level function bindings) can plausibly be regarded as local bindings, but that for the purposes of coding, we are ignoring these. This applies to functions declared in the current scope (as in problem 7), but not to functions passed as parameters (as in problem 8).
BND/NOLOCALS(23)	The program contains at least one local variable that does not appear in the solution. Note that some solutions may show the parameter bindings as the lowest part of the caller's frame, above the saved frame pointer. This corresponds to certain ABI conventions, and should probably not be considered wrong.
BND/NOCLOSURES	Parameter bindings do occur in the stack, but not those whose value is a closure. Note: This implies BND/PARAMS.

Handling of closures:	
CLO/OPAQUE(8)	Closures are written as opaque structures, such as `g*` or `f(3)` or `g(def)` or an arrow to a function definition location or similar.
CLO/CLEAR(8)	Closures are written as structures with contents, ideally including source, parameter names, and environments. Including a name and the closure's environment is enough, though.

Figure 10: Rubric for Pre-Test, Part 2. Continued from fig. 9.

Scorability:

SCORABLE	This solution is sufficiently correct to be assigned scores. Specifically, it differs from the correct solution in ways that are attributable to comprehensible errors. Solutions that have stack frames that seem unrelated to the problem, or supply many disconnected stacks, or are entirely blank, should not receive marks in this box, or in any other category. Solutions that don't get a mark in this box should have some kind of comment indicating the nature of the submitted solution.

Use of Snap*!*:

SNAP	This solution uses Snap*!*, submitting either a screenshot of a solution using the blocks or just the XML file that is generated by Snap.

Notions of the stack:

S/BYTES	*Unchanged.*
S/LOGICAL	The stack is a logical part of a step in a notional machine. Frames are clearly separated, probably have arrows connecting them, contexts may be represented as source code with holes, environment values are definitely labeled. Note that solutions using Snap*!* will always have this property.

Behavior of the stack:

This section did not change.

Handling of environments:

E/SEPARATE	This solution separates environments from stack frames. Solutions that place an box around the environment should receive this mark, even if that box is drawn inside of the stack frame box in a non-snap solution. Solutions that use Snap*!* will always have this property.
E/CHAIN	This solution correctly links the environment of a stack frame to that of the lexically enclosing frame at least once.
E/BADCHAIN	This solution incorrectly links the environment of a stack frame to that of the lexically enclosing frame, or is missing a link in a non-top-level frame. Note that both E/CHAIN and E/BADCHAIN can be assigned to a single solution.

Figure 11: Rubric for Post-Test, Part 1. Continues in fig. 12.

Handling of contexts:

C/ADDR	*Unchanged.*
C/CTXT	*Unchanged.*
C/NOHOLE	At least one of this solution's contexts do not contain a hole. If this box is checked, C/CTXT should also be checked.
C/SUBRIGHT	At least one variable to the right of the hole has been replaced with a value. If this box is checked, C/NOHOLE must not be checked, and C/CTXT must be checked.
C/OTHER	The context contains a different error. Certain clerical errors should not be included here; the reduction order of a three-armed plus is not important, a minor arithmetic error like adding 3 and 4 to get 8 is not important. Also, ignore errors related to the omission of earlier or later lines in a multi-line context. When this box is checked, make a note to indicate the nature of the error.
C/NONE	No context: the solution omits any reference to context, containing only bindings and/or frame pointers. For solutions using Snap*!*, this should be assigned for completely blank context slots.

Handling of bindings, both parameters and local variables:

BND/PARAMS	*Unchanged.*
BND/BADPARAMS	At least one parameter binding is missing or there is a parameter binding that does not belong (either because it is in the wrong place or it is just extra). This includes drawn solutions that show the call as e.g. `f(2,4)` (that is, where the parameter values can be deduced from the drawing but are not shown as taking independent space in the stack). This also includes drawn solutions where the student has performed substitution on a copied body of the called function.
BND/LOCALS(1)	The problem contains at least one local binding that appears in the solution in the correct stack frame or environment. This does not include top-level bindings.
BND/BADLOCALS(1)	*Unchanged.* (This is just BND/NOLOCALS renamed.)
BND/TOPLEVELS(3)	The problem contains at least one top-level binding that appears in the solution in the correct place.
BND/BADTOPLEVELS(3)	The problem is missing at least one top-level binding. Note further that function bindings (including top-level function bindings) can plausibly be regarded as local bindings, but that for the purposes of coding, we are ignoring these.

Figure 12: Rubric for Post-Test, Part 2. Continued from fig. 11.