Combining Interactive and Whole-Program Editing with Repartee

Luna Phipps-Costin[1], Michael MacLeod[2], Alex Vo[3], Tiffany Nguyen[4], Joe Gibbs Politz[5], Shriram Krishnamurthi[6] and Benjamin S. Lerner[7]

[1]lphippscosti@umass.edu; UMass Amherst
[2]mmmacleo@ucsd.edu; UC San Diego
[3]amv036@ucsd.edu; UC San Diego
[4]ttn014@ucsd.edu; UC San Diego
[5]jpolitz@eng.ucsd.edu; UC San Diego
[6]shriram@brown.edu; Brown University
[7]blerner@ccs.neu.edu; Northeastern University

Abstract

Interactive evaluation with a REPL (Read-Eval-Print Loop) is a feature of many programming environments, especially in environments for teaching programming. However, REPLs do have confusions and stumbles for beginners related to navigating between programs and interactive evaluation. We identify several specific weaknesses of REPLs with a worked example from an existing programming environment in active use, distilled from our experience with novices. We then present an updated programming environment that mitigates these weaknesses by combining the program editor and REPL, so the user can benefit from the best of both. *Keywords*: Programming environment, REPL, introductory programming.

1 Introduction

In the 1960s, John McCarthy and his LISP collaborators created the Read-Eval-Print Loop (REPL) [1], which provided an entirely new model of interacting with programs. Rather than writing an entire program and waiting for its results, programmers got an interactive interface to program evaluation. Incremental program snippets could be sent to the programming language runtime with immediate responses and incremental evaluation added to a running program state.

Since then, this basic idea has spread and been iterated on. Languages that shipped without REPLs were forced to add them, like Java with `jshell`. The browser developer console, popularized by Firebug,[1] was broadly deployed across all major browsers. Computational notebooks follow in a similar spirit, also being popular for languages like Python.

The immediacy of responses from REPLs and similar tools is critical to many introductory curricula. First, immediately juxtaposing results with the expression that produced them in a shared space supports students' association of expressions and values while avoiding the need to context-switch between code and output interfaces. Second, encouraging quick feedback from the evaluator, one REPL entry at a time, reduces the amount of code that could be to blame when a student encounters an error. These two properties are usefully described as spatial and temporal immediacy by Ungar et al. [2], and "support incremental running and testing with immediate feedback" with a "dialog mode" are explicit suggestions from Pane and Myers [3].

In 2013, several of the authors started implementing a new programming language, Pyret, designed for education. A REPL featured prominently in its programming environment's design. The environment, CPO (`code.pyret.org`), has a REPL that mimics DrRacket with an area for writing program definitions (called the definitions area), and an interactive interface that renders results from individual entries (called the interactions area) [4]. This REPL already incorporates some lessons learned from years of deployment of DrRacket, which include rendering image values directly in the interface, built-in support for reporting test case results, and more.
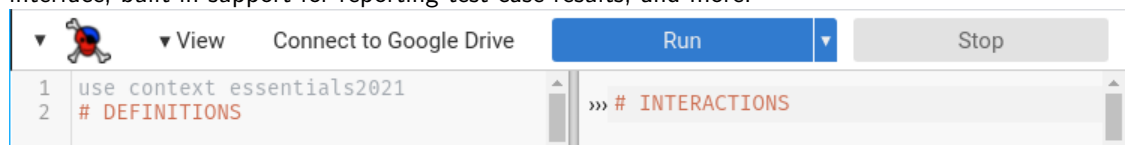
---

1   https://getfirebug.com/

Over the past 8 years, Pyret has been used by tens of thousands of students in many contexts from secondary school physics classrooms to upper division collegiate electives. During this time, several of the authors of this paper have worked with hundreds of teachers, many with no programming background, to help them use and teach Pyret using CPO. Based on our repeated firsthand observations of beginners' first interactions with CPO's REPL-based environment, we believe we've found significant opportunities for improvement.

There are two contributions of this paper:

- A distillation of generalizable shortcomings in CPO, told as a series of example interactions representative of what we see with learners in practice.
- A new design for a REPL-like system that addresses the issues we identify.

We see the primary shortcomings of CPO's environment as stemming from the inability to edit past entries, the lack of a resulting correct, persistable program, and hidden state between definitions and interactions. We believe these problems generalize to other REPL environments.

We proceed by first walking through a sample interaction a user might have with CPO, highlighting specifically where existing REPL designs fall short. We then discuss our prototype response and some of its consequences. All of the examples in this paper can be run by the reader, with CPO at code.pyret.org and REPARTEE at https://pyret-anchor.s3.amazonaws.com/anchor/index.html.

2 Using the CPO REPL

Many Pyret-based curricula teach function composition, an important early learning outcome. Image composition tasks serve this learning outcome, can be quite short, and provide immediate visual feedback to check correctness. As a standard early task, a student may select an image that they connect with—a logo, an emoji, a flag, etc.—to create with a program.
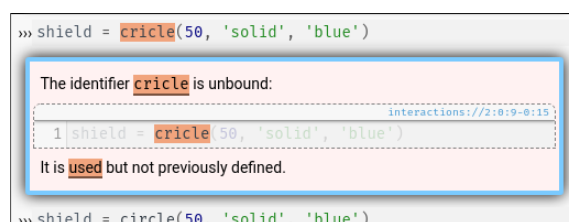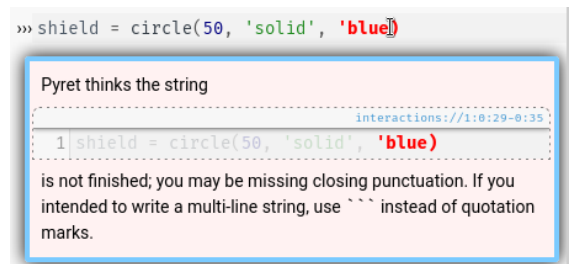
We proceed with an example of the common case where students have been encouraged to start in the interactions area—the REPL—of CPO. Our hypothetical student has used the image library functions in a basic way and is now learning to compose them. They have chosen to create Captain America's shield.[2] This is the program the student will eventually write and its correct output:

```
r1 = circle(50, 'solid', 'red')
r2 = overlay(circle(40, 'solid', 'white'), r1)
r3 = overlay(circle(30, 'solid', 'red'), r2)
r4 = overlay(circle(20, 'solid', 'blue'), r3)
shield = overlay(star(20, 'solid', 'white'), r4)
```


The student starts by writing shield = circle(50, 'solid', 'blue) (which has a syntax error, missing a closing quote) to form the outer ring of the shield, then presses Enter. The line is immediately given to the evaluator, which produces a syntax error. The error helpfully highlights the missing quote, so the student clicks right before the parenthesis to try to edit the entry they just wrote. Frustratingly, they cannot; past entries in interactions are immutable and the text area cannot be edited.

They didn't quite catch what the teacher meant by "pressing the up button to get the last thing back," and they don't immediately think to copy-paste, so they go ahead to re-type their mistaken line: shield = cricle(50, 'solid', 'blue') (with a misspelled circle). They press Enter, and are greeted with another

---

2 A video showing this interaction from beginning to end is at https://drive.google.com/file/d/1fposbW87BPfHzNdKSLTVJPjTYmowmlMI/view?usp=sharing.

error.   81

Had their addition of the quote unveiled   82
some other problem? No, they realize, it was   83
a different typo from re-typing the line. Correct-   84
ing the line once more, no error is output this   85
time, but rather nothing is output at all. Be-   86
cause their line defined a new variable, there's no output. At this point, they give up and raise their   87
hand for help from their teacher, who reminds them that typing `shield` at the prompt gives them   88
their blue circle. We note at this point that learners arriving at this step have repeatedly expressed   89
frustration that multiple errors persist onscreen, both because it feels like a repeated judgment from   90
the environment and because it takes up so much space. Students say things like, "how can I make   91
the errors go away?"   92

Satisfied with their circle, they move on.   93
They create a smaller white circle and overlay   94
it on the blue circle, creating a blue ring and set-   95
ting up for a white ring. Not wanting to think of   96
a new name, since they're still building a shield,   97
they name the result `shield`. Having learned   98
to check their typing carefully, they confidently   99
press Enter, and are confronted with an error   100
that they cannot redefine `shield`.[3] Having been   101
told by an observing teacher about the up-arrow   102
key, they press up-arrow and choose the name   103
`ring2`. They realize the first circle really ought to be named `ring1`, but it's too mixed up in the   104
history with errors and the `shield` they typed just to see their work, and it would visually follow   105
`ring2`, so they'd rather not bother.   106

Wanting to check their work, the student   107
overlays the remaining rings and white star to   108
their partial shield one-by-one, naming each   109
stage incrementally. Finally they're presented   110
with their shield, but something looks off. Check-   111
ing the image, they notice that the outer circle   112
should be red, but in their shield it's blue. They   113
defined the outer circle way at the very top of in-   114
teractions. They realize they will have to go find   115
the large circle to edit it (or re-create it entirely).   116
Then, they must update every other line of the program again to reference the new circle. Their heart   117
sinks.   118

This time, though, they've grown tired of the cruelty of the REPL. Having learned that the   119
definitions area can be used for things that might change in the future, and worried they might have   120
to change another thing about this ring, they type their new red circle in the definitions area. They   121
name it `r1` so that each stage can have consistent names, and press Enter: 122

   123
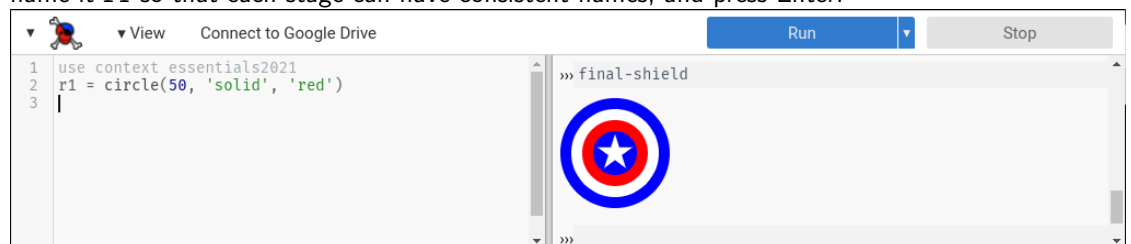
The Enter keypress inserts a newline into the definitions area, which is jarring to the student   124

---

3 This reflects a design choice of these curricula to use functional binding with =, which disallows rebinding in Pyret, rather than introduce mutation early.

because they've been using Enter to run their programs. But the definitions area is a whole-program text editor, so the Run button runs the definitions, not the Enter key. Focusing back on interactions, they scroll through their up-arrow history to find `ring2`, edit its use of `shield` to `r1`, and its name to `r2`. They press Enter again, and receive an error that `r1` is unbound.
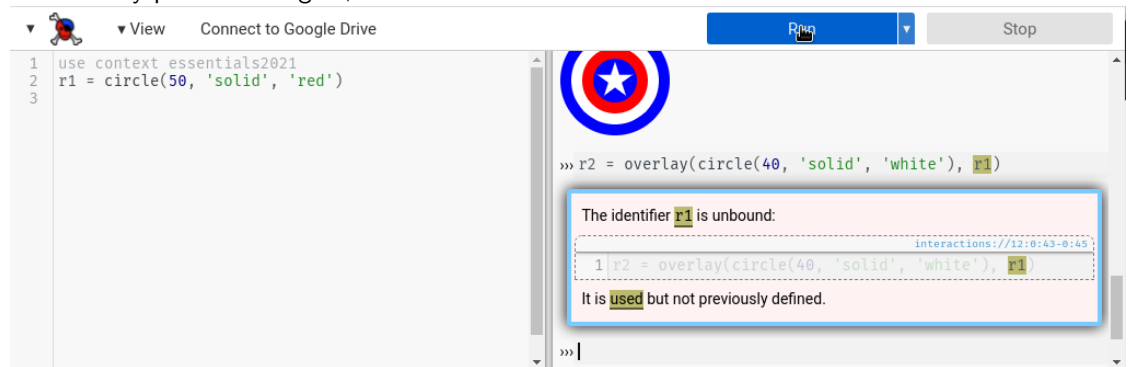
They're confused, since they can see r1 defined on the screen, over in the definitions area! Realizing their mistake, they click the run button, and look in horror as the entire program they crafted in the interactions window suddenly disappears:
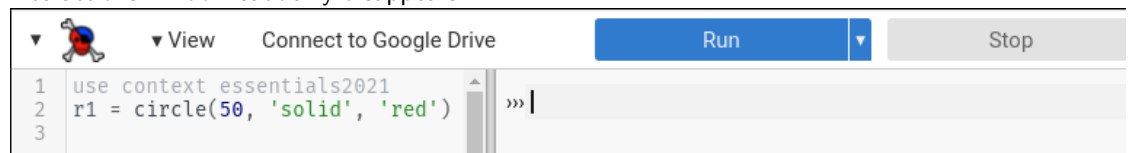
This clearing is an intentional design decision, borrowed from DrRacket, where re-running the program definitions clears any state defined in the interactions area. This is because any state defined in the interactions area could now be inconsistent with the updated version of the program. This idea, called the transparent REPL, was a refinement on original REPL designs that could confuse students with hidden state across runs [4].

Fortunately, the interactions history is saved, even though it is hidden, so using the up-arrow they're able to recall the r2 that had just errored:

Now in the habit of pressing run (the effect is worse if definitions has been run a few times or if the keyboard shortcut for running, Ctrl-Enter, has been learned), they click the Run button instead of pressing Enter, clearing the entry they just recalled.
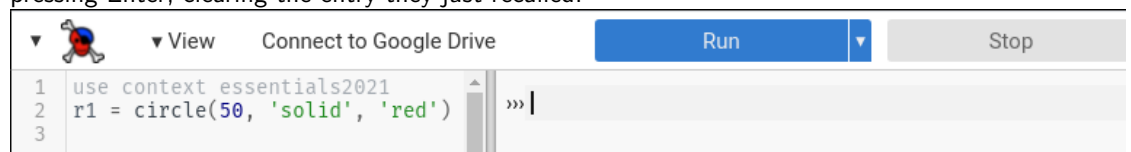
The student must keep in their head that to run *definitions*, they must press *Run*, but to run *interactions*, they must press *Enter*. The student is then able to go through the motions of recalling and editing each previous line, arriving at the final version of their chosen image.

However, even at this point they may not be "done." In order to *save* their program, they have to put any code they want to persist into the definitions area, as that is where persistent *programs* go, as opposed to just interactions. So they need to remember (or find time as seconds tick off before the lunch bell) to copy these interactions to definitions and save.

We've just seen an example of a frustrating interaction we've observed many times. Admittedly, some of these frustrations are actually instructive. First, it's relevant to function composition and programming learning outcomes to understand that each time `overlay` is used a new value is created. Second, reading error messages and fixing mistakes is relevant for learning outcomes about attention to detail and understanding notation.

However, the definitions/interactions distinction and the immutable history of interactions cause

serious *incidental* frustrations that don't correspond to any learning outcome that we care about. ¹⁵⁹ These issues are what prompted us to explore other interface decisions that could support the learning ¹⁶⁰ outcomes we care about while alleviating the unnecessary frustrations. ¹⁶¹

3 Solution Context: Audience, Goals, and Constraints

In response to the above confusions and others we've observed, we've designed a new REPL-like system. Before we describe it, we first present the context of its use.[4]

Audience   Pyret's primary audiences are (1) teachers and students in high school courses and (2) first-year undergraduate students in computing majors, especially focused on data-centric introductory programming [5]. In particular, the high school courses are predominantly *non*-computing courses, like algebra, statistics, or physics.

Goals   We are focused on the experience of the first 20-40 hours of programming. This is the amount of programming in many professional development workshops with teachers, and indeed a close match to the total number of hours of classroom instruction using computing in some of our curricula. Naturally, undergraduate programmers do more programming than this in a semester of a programming course. Nevertheless, a good early experience is still important to avoid discouraging students and, in curricula where students have freedom to choose courses, to keep them from dropping prematurely—especially if it has a negative impact on diversity [6].

Constraints   Finally, we are constrained in several ways. First, there are many existing users of Pyret, and too radical a departure risks losing valuable community around the language; we want to produce a new experience for writing Pyret programs that is mostly backwards-compatible with existing curricula, not a new programming language and environment. Second, we are, for the purposes of this work, committed to text-based programming. We view a block-based editor as primarily complementary to this work, as we mention in section 7.

4 From REPL to REPARTEE

We proceed by demonstrating the same example in our updated interface, dubbed REPARTEE (Read, Eval, Print, and Respond to Each Edit). REPARTEE is designed around two main ideas:

1. Many languages give the programmer two ways to write programs: an editor and a REPL, on CPO called definitions and interactions. They have complementary features, and it takes significant effort to learn when to use one or the other. Moreover, when the programmer (especially a novice) uses the wrong one, there can be serious consequences: losing all state in interactions or not getting immediate feedback in definitions. REPARTEE blends the REPL and the editor to give the advantages of both.[5]

2. We live in a world awash with messaging, especially amongst our audience. Messaging applications like Slack, iMessage, WhatsApp, Discord, Telegram, Signal, Facebook Messenger, Instagram/Twitter DMs, and more are a part of daily life. REPARTEE takes on the look and feel of such applications to leverage familiarity with those environments. This metaphor also suggests affordances, like those for editing already-sent messages, that are present in modern messaging but not in existing REPL interfaces.

We now consider what the same student's experience would have been had they taken the same steps using REPARTEE.[6]

They begin their program typing in a large text entry box at the *bottom* of the page, where new messages are written in messaging apps. First, being alternate-universe twins, they make the mistake

---

4 We believe the issues presented generalize to many REPL systems, but we focus on the CPO use case to keep the presentation concrete.

5 A reader familiar with computational notebooks might wonder if this is a problem that has been solved already; we highlight some differences throughout the coming example and in section 6.2.

6 A video showing this interaction from beginning to end is at https://drive.google.com/file/d/1FowaQ-4YUKRw7kPH GlPnQeTIUhxXxQK-/view

of forgetting a quote. They have two choices to run their program – they can press the Send button, or use the keyboard shortcut Ctrl+Enter (hinted from below the prompt). Either has exactly the same meaning.


Sending the entry results in it appearing as a sent chat message, with an immediate response indicating the error:
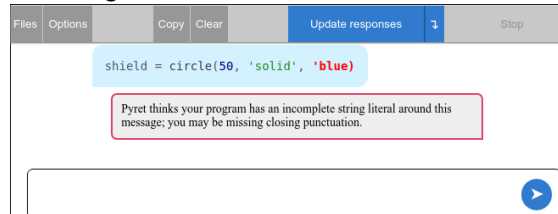

An observant reader may have noticed the error message is slightly different than in CPO. This is partially due to internal implementation details between the two systems, but also since the error message will always be rendered next to offending code, error reports that duplicate fine references to code are less necessary.

In CPO, the student tried to edit the entry, but couldn't. They had to somehow recall (using a keyboard shortcut or copy-and-paste) or rewrite the entry, leading to even more mistakes. However, in REPARTEE, as in many messaging apps, the student is able to directly edit the previous entry. They click on the relevant area of the previous entry, and enter a quote. Immediately upon typing the quote character, both the entry and the result's borders turn dashed.
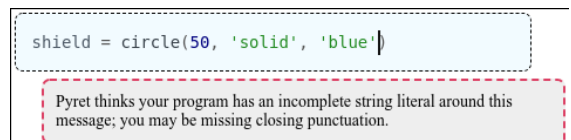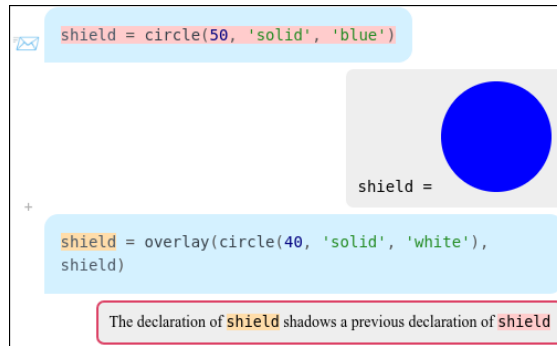

The entry is dashed to indicate it has been edited; the result is dashed to indicate it *may* no longer display a response that corresponds to the (edited) program. This addresses one immediate concern some users have with computational notebooks – it's not clear if the current state of the text reflects the current output.[7] The student presses Ctrl-Enter and the program is run again.
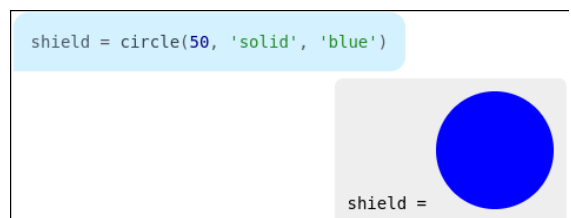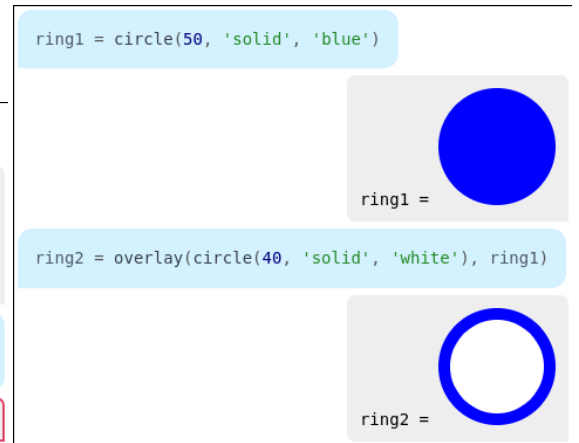

REPARTEE displays the values of top-level bindings, so the student is able to see their large blue circle and confirm they're on the right track without cluttering their history. When the student makes the same shadowing error on `shield` that they did with CPO, they can easily proactively rename the first to `ring1` since (1) the interface allows in-place editing of past entries and (2) there aren't multiple error messages taking up vertical space, so the definition is easily accessible:
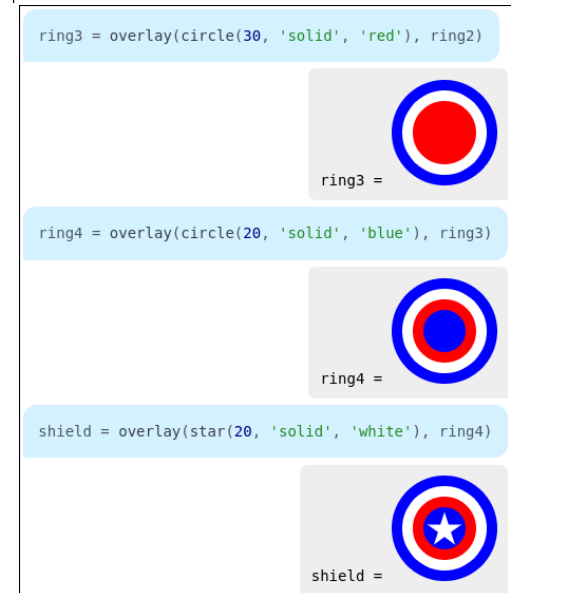
---

7 https://conferences.oreilly.com/jupyter/jup-ny/public/schedule/detail/68282.html

```
shield = circle(50, 'solid', 'blue')
```

shield =

```
shield = overlay(circle(40, 'solid', 'white'),
  shield)
```

The declaration of **shield** shadows a previous declaration of **shield**

The student writes the rest of their program without issue, and, like with CPO, is concerned as they realize it has a miscolored ring.

```
ring1 = circle(50, 'solid', 'blue')
```

ring1 =

```
ring2 = overlay(circle(40, 'solid', 'white'), ring1)
```

ring2 =

234

```
ring3 = overlay(circle(30, 'solid', 'red'), ring2)
```

ring3 =

```
ring4 = overlay(circle(20, 'solid', 'blue'), ring3)
```

ring4 =

235

```
shield = overlay(star(20, 'solid', 'white'), ring4)
```
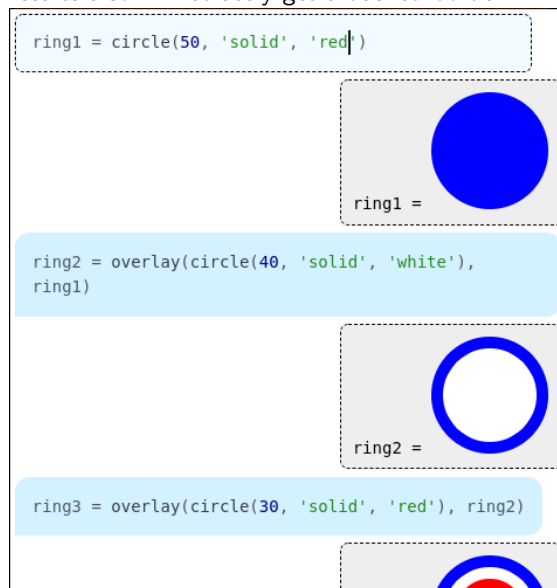
shield =

Confident in their previous attempts at editing past entries, they click on the entry with the definition   236
for `ring1` and edit the text, changing `'blue'` to `'red'`. The entry's border immediately becomes   237
dashed, indicating the program has changed. In addition, the entry's result and all of the following   238
results also immediately get a dashed border: 239

```
ring1 = circle(50, 'solid', 'red')
```

ring1 =

```
ring2 = overlay(circle(40, 'solid', 'white'),
  ring1)
```

ring2 =

```
ring3 = overlay(circle(30, 'solid', 'red'), ring2)
```
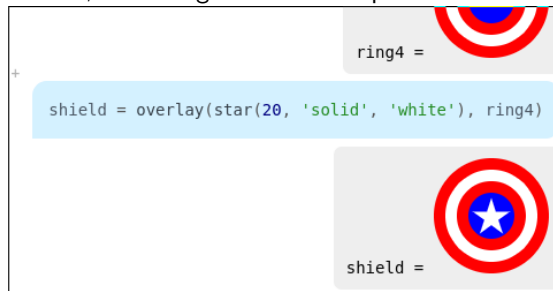
240

Then they press Ctrl-Enter to run the program. After they do, each entry is run in order from the   241
beginning. The red circle propagates one-by-one through each of their results, and each dashed line   242

---

is removed, indicating the result is up-to-date.[8] The final entry's result shows the correct shield:

The dashed borders on results indicate that they display results that may no longer correspond to their entry's value because the program has changed: in the example above, each result contained the erroneous blue ring until the program was re-run.[9] This is a cue to both the current user *and to any instructor arriving to look at their screen* that a run will be needed to reach a consistent view of the program's behavior. This avoids a common complaint with computational notebooks [7] that it can be difficult or impossible to determine this at a glance.

In addition, since the series of messages is not intended to be transient unlike the interactions area of CPO (although it can be transient just as easily), there is no extra copying or text-management process for saving the program. The entire program has been saved the entire time, so there is no need for the student to consider extra steps to save their work before the end of class.

We've just seen an example of an updated interface that we believe improves the experience compared to the current CPO. This student still confronted error messages, and still had to fix their program. They still had to make decisions about how to structure their use of functions, and were free to make mistakes in doing so. These are all related to learning outcomes for composition and programming. They *didn't* have to confront issues related to managing text in two very different modalities, experience a jarring failed attempt to edit their program, or have to redo any work to make the necessary change, all of which were unnecessary frustrations with the current CPO.

5 Repartee **vs Editor Plus REPL**

The concrete example above shows some specific ways Repartee improves on the experience of the current editor and REPL. A student can edit previous entries, which is often their inclination upon seeing their first error in interactions. They get the benefits of seeing immediate responses to entries, can more easily maintain an error-free program, and see consistent updates to values when they are referenced later. The result is a working, saved program—not a scratchpad of interactions—that has been developed incrementally with many of the benefits of the REPL. Repartee provides more verbose feedback acknowledging definitions and showing their values, ensuring students see the results of their work. Repartee unifies the editor (definitions) and the REPL (interactions), having only one area which is at once editable and segmented and with only one concept of run and submit. This eliminates state associated with run order between areas, and Repartee makes the remaining state of edited entries and outdated results visible.

Repartee maintains transparency by running the entire program from the beginning on every run, avoiding any hidden state from past runs. As an optimization, when a new entry is added to the end, Repartee runs only that entry in the existing context, and the result is the same as if the program had been run just once including the final entry.[10] The program can still be run from the beginning using the Update Responses button or the global keyboard shortcut Ctrl+Enter.

---

8 This propagation would not happen in the default settings of e.g. Jupyter Notebook.

9 This is a departure from most messaging interfaces, which typically only display an edited marker on the edited message itself, and not on the responses which may have critically changed or lost context. We wonder if this idea could be useful in the messaging context, and a similar metaphor could be used if Twitter adds the sought-after edit button.

10 In deterministic programs with no I/O (such as those produced by our audience), this and some other similar optimizations are also equivalent to running the program from the beginning.

6 Related Work

6.1 Compared to Live Programming Environments

Live programming [8] environments re-run whole programs with high frequency, as often as each edit. For example, Omnicode [9] is a live system targeted at novices that continuously re-renders values as the program changes, with similar goals to our example of changing the color of the shield's ring. Hundhausen and Brown describe a live strategy for algorithmic code with some evidence that the quick feedback loop helped student learning [10]. We don't categorize REPARTEE as "live," and it is a deliberate choice. Pedagogically we often highlight the moment we choose to run the program as an opportunity to stop, think, and predict output. Indeed, the model of REPARTEE (and REPLs) encourages creating discrete runnable units where the user explicitly chooses the act of running.

Some aspects of REPARTEE draw direct inspiration from the Desmos Graphing Calculator [11], a kind of live programming environment for math. It has support for function definitions and function composition in mathematical notation, where each definition goes in a separate cell. Errors are rendered adjacent to cells that cause the errors and cells can be freely edited. Unlike REPARTEE, Desmos focuses on a single graphical output where all definitions are shown, which fits its coordinate-plane graphing use case well in contrast to a broader set of uses for Pyret and REPARTEE.

6.2 Compared to Notebooks

Computational notebooks [12]–[14] are a popular tool both in education and for professionals. They have a key similarity with REPARTEE: separate text entry areas that are juxtaposed with results from those computations. The evaluation models of notebooks and REPARTEE are quite different, however. In notebooks, cells can be run out of order [7], so the state of results is dependent on both the text of the entries and the entire history of the user's decisions about what to run. In REPARTEE, the up-to-date state of all entries, run from the beginning, is shown after each run. If an edit or error happens, results and entries are invalidated appropriately. It's important to note that while it may take some time for a learner to internalize these cues, an instructor helping the student can make use of them. In notebooks, no invalidation cues exist on cells or on results.

We implemented REPARTEE atop the JavaScript-based runtime for Pyret. We believe that interfaces similar to REPARTEE could be built atop the infrastructure of computational notebooks.

Similar Notebook Variants Other systems have provided versions of notebooks that are targeted at alleviating some of the issues with out-of-order execution, which REPARTEE also avoids.

Bostock describes Observable, which is a nearly-JavaScript programming language and notebook service where a dataflow analysis is used to detect and re-run cells that depend on updated values [15]. Similarly, Nodebook is an plugin for Jupyter Notebooks that enforces that cells run in a dependency order based on their use of variables in other cells [16]. Observable allows definitions to appear after their use, resolving the order of evaluation by using dependencies between cells, which REPARTEE disallows because it is focused on a close match to a straight-line, eager semantics. Both Observable and Nodebook don't prevent running cells if a dependent has an error, which can lead to a cascade of error messages on later cells. In contrast, REPARTEE stops evaluation on the first error, which is where early programmers should focus their effort.

Observable and Nodebook may only re-run a subset of cells, while REPARTEE always re-runs the whole program from the top on an edit. Re-running fewer cells while maintaining useful reporting to the user about what is out of date is a fruitful avenue of future work for REPARTEE. Since we want to give quick and sound information about what is potentially invalidated on edit, REPARTEE is conservative and invalidates all future results on edit without trying to reinterpret the dependencies between cells.

6.3 Other Related Novice Programming Environments

There are many programming environments that target novices; we highlight a few that are especially related to REPARTEE here. Lee describes Gidget, which uses a personfied conversational interface to aid students in debugging [17] and concludes that it helps novice programmers respond to feedback

effectively. Repartee is designed to look like a messaging application, both to leverage students' likely experience with similar interfaces and to evoke a personal message-response understanding of the evaluator.

Other environments, like Snap! [18] and Scratch [19], are targeted at the same initial programming experience as we are with Repartee. These environments serve different learning goals than the curricula Repartee supports. Function composition, for example, is not an early learning outcome in Snap! and Scratch tutorials, which are more focused on sequential composition of statements that affect a user-visible graphical "stage." These different early programming models serve users in different ways.

7 Future Work

There are several areas of future work we are interested in exploring with Repartee.

First and most directly, we will gradually start using Repartee with users to gather their feedback and our own observations of their initial experiences. This will help us understand how our intended improvements change the kinds of questions, frustrations, and confusions first-time users have, and iterate further.[11]

Second, there are new user interface options available that we have noticed. One example is that there is no reason that each entry needs to be *authored* in the same way. That is, some entries could be authored with a block-based editor, some could be plain text, others could be markdown-formatted comments (similar to a notebook), and others could have custom scaffolding for building e.g. a function definition versus a set of test cases. We're interested in how the separation of entries provides this affordance, in contrast to the definitions area which is more "all-or-nothing" in terms of swapping in a new style of editing, and also in contrast to the interactions area where more heavyweight editing support for a temporary entry has felt mismatched.

Third, the Repartee interface no longer needs as much horizontal space, since there aren't separate panes for definitions and interactions. We're interested in using the left and right column space for showing useful auxliary information, like documentation or program traces, which can more naturally show up adjacent to the program rather than requiring a separate pane or window.

Finally, there is an interesting semantic consequence of programs written with Repartee. Since each entry has access to the context aggregated from all preceding entries, we can proceed in the style of Petricek's Histogram [20] and give richer access to type information calculated from values in early entries. As a concrete example, the environment could provide static information to later entries (e.g. autocomplete, or before-run errors) about tables dynamically loaded from a Google Sheet in an earlier entry. While this context is technically also available at interactions, it is not in the evaluation model of definitions, where the whole program is type-checked before any dynamic loading happens.

References

[1] J. McCarthy, R. Brayton, D. Edwards, P. Fox, L. Hodes, D. Luckham, K. Maling, D. Park, and S. Russell, "Lisp i programmers manual," Artificial Intelligence Group, MIT Computation Center and Research Laboratory, Tech. Rep., 1960.

[2] D. Ungar, H. Lieberman, and C. Fry, "Debugging and the experience of immediacy," *Communications of the ACM*, vol. 40, no. 4, pp. 38–43, Apr. 1997.

[3] J. F. Pane and B. A. Myers, "Usability Issues in the Design of Novice Programming Systems," Carnegie Mellon University Human-Computer Interaction, Tech. Rep., 1996.

[4] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen, "DrScheme: A programming environment for scheme," *Journal of Functional Programming*, vol. 12, 2002.

[5] K. Fisler, S. Krishnamurthi, B. S. Lerner, and J. G. Politz, *A Data-Centric Introduction to Computing*, Accessed September 14, 2021, 2021. [Online]. Available: https://www.dcic-world.org.

---

11 Repartee as presented here already reflects several iterations based on design and feasibility checks we have already done.

[6]   A. Lishinski and A. Yadav, "Self-Evaluation Interventions: Impact on Self-Efficacy and Performance in Introductory Programming," *ACM Transactions on Computing Education*, vol. 21, no. 3, Jun. 2021.

[7]   S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, and T. Barik, "What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 2020.

[8]   *Proceedings of the 1st International Workshop on Live Programming*. 2013.

[9]   H. Kang and P. J. Guo, "Omnicode: A novice-oriented live programming environment with always-on run-time value visualizations," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, 2017.

[10] C. D. Hundhausen and J. L. Brown, "What you see is what you code: A "live" algorithm development and visualization environment for novice learners," *Journal of Visual Languages & Computing*, vol. 18, no. 1, 2007.

[11] Desmos, *Desmos Graphing Calculator*, Accessed September 14, 2021, 2021. [Online]. Available: https://www.desmos.com/calculator.

[12] D. S. Arnon, "Workshop on Environments for Computational Mathematics," *SIGGRAPH Computer Graphics*, vol. 22, no. 1, Feb. 1988.

[13] S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer (2nd Ed.)* USA: Addison Wesley Longman Publishing Co., Inc., 1991.

[14] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, *et al.*, *Jupyter Notebooks-a publishing format for reproducible computational workflows*. 2016.

[15] M. Bostock, *Observable's not javascript*, Accessed September 10, 2021, 2019. [Online]. Available: https://observablehq.com/@observablehq/observables-not-javascript.

[16] K. Zielnicki, *Nodebook*, Accessed September 10, 2021, 2017. [Online]. Available: https://multithreaded.stitchfix.com/blog/2017/07/26/nodebook/.

[17] M. J. Lee, "Teaching and engaging with debugging puzzles," Ph.D. dissertation, University of Washington, Sep. 2015.

[18] B. R. i Carrasquer and Snap! Team, *The Snap! Programming System*, Accessed September 13, 2021, 2019. [Online]. Available: https://link.springer.com/referenceworkentry/10.1007/978-3-319-60013-0_28-2.

[19] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch Programming Language and Environment," *ACM Transactions on Computing Education*, vol. 10, no. 4, Nov. 2010.

[20] T. Petricek, *Histogram: You have to know the past to understand the present*, Accessed September 10, 2021, 2017. [Online]. Available: http://tomasp.net/histogram/.