Teaching Programming Languages by Experimental and Adversarial Thinking*

Justin Pombrio[1], Shriram Krishnamurthi[2], and Kathi Fisler[3]

1 **Computer Science, Brown University, Providence, RI, USA**
2 **Computer Science, Brown University, Providence, RI, USA**
sk@cs.brown.edu
3 **Computer Science, WPI, Worcester, MA, USA**

Abstract

We present a new approach to teaching programming language courses. Its essence is to view programming language learning as a natural science activity, where students probe languages experimentally to understand both the normal and extreme behaviors of their features. This has natural parallels to the "security mindset" of computer security, with languages taking the place of servers and other systems. The approach is modular (with minimal dependencies), incremental (it can be introduced slowly into existing classes), interoperable (it does not need to push out other, existing methods), and complementary (since it introduces a new mode of thinking).

1998 ACM Subject Classification D.3.3 Language Constructs and Features, K.3.2 Computer and Information Science Education

Keywords and phrases mystery languages, interpreters, paradigms, education

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.13

1 An Enumeration of Principles

What do we want students to get out of a programming languages course? There are many things we might hope for, some of which include:

- The ability to distinguish syntax from semantics: to understand that a syntax is not "deterministic", in that one syntax can have many different meanings, and these differences of meaning can trip up the unwary programmer.
- The ability to look past superficial labels (such as "object-orientation") to understand features in depth, including the profound variations between their different realizations. (For instance, even with classes and inheritance, single-, multiple-, and mixin-inheritance result in very different languages and corresponding programming styles).
- The ability to discern good design from bad. This is, of course, a deeply personal matter of judgment: one person's beauty is another's monkey-patch. A good instructor presumably does not only impose their own world-view but also helps students understand the design trade-offs between different approaches, and also provides some historical background that helps students appreciate how current designs came to be.
- The ability to learn new programming languages. Especially in an era where we see a kind of "Cambrian explosion" of languages from both academia and industry, it is essential for students to have skills to pick up new languages quickly and accurately.

* This work was partially supported by the US NSF.

These are some of the principles *we*, the authors, want students to learn, and we constantly re-design our courses to help students get closer to learning these ideas. This paper describes our latest and most radical re-design to get at the heart of this learning.

In this document, we will use the term *programming language* with a certain meaning: with an emphasis on *programming*. That is, we don't mean a formal calculus. We mean an actual system that one can run and whose programs can actually do something of interest to people other than pure programming-language researchers.

2 How We Teach and Learn About Languages

How do courses address these principles? Most undergraduate ("bachelor's") courses that focus on language principles – rather than implementation methods – seem to fall into two major schools. One approach subdivides languages by *paradigm* (though some authors have argued that the very concept of a paradigm is flawed [7]), and teaches students about variations within the paradigm as well as differences between paradigms. Some of the most widely-used programming languages books (like that by Sebesta [9]) are organized along these lines. Another view (formerly embraced by the present authors) is that one learns best by "teaching" a language to the patient but unforgiving computer, in the form of *definitional interpreters* [8]. This approach is also widely taught [1, 4, 6, 5].

We are unconvinced that either approach addresses our four principles particularly well. Though some of these can be addressed, it would be at a significant cost. For instance, most interpreter-based courses rarely implement multiple semantics for one syntactic feature (usually excepting function application or, in special cases, garbage collection [2]). They *could*, but even a high-level implementation can take a lot of time, and simply producing an interpreter does not by itself force reflection on programming *in* the implemented languages.

In contrast, it is worth pausing to ask the following question:

> What do *you*, dear reader, do when confronted with a new programming language?

For instance, consider one of the major recent languages: industrial ones like TypeScript, Flow, Hack, Swift, or Dart; or more academic ones like Agda, Idris, Liquid Haskell, or Rosette. Imagine you needed to learn one or more of them. Would you break it down by paradigms alone? Do you imagine writing a definitional interpreter? Or would you dive in and start writing programs to explore the space of the language? Would you look up a few examples, try them out, then embark on exploring the state space – like an adversarial thinker does – by asking, "Hmm, I wonder what happens if I tweak *this* to say *that* instead?"

3 Languages as Natural Science Artifacts

The reality of any programming language is that it is a complex ecosystem of parts that interlock in interesting ways, most of them at best poorly specified. They are rarely accompanied by a formal semantics. Even when a semantics exists, it may not cover the whole language, including the parts that programmers need to "get things done". Much of the semantics may lie instead in natural language prose documentation. Some of it may only be expressed in the language implementation's test suites.

Of course, even when a language is completely formally described, and when a programmer can read and comprehend the semantics, this is of only so much use. There is a large gap between the semantics itself and its *consequences*: for instance, the difference between eager and lazy evaluation strategies can be summed up completely in a few characters of the

right semantics specification system, but the consequences of these differences – the different data structures they enable, the different proof methodologies they require, the different time/space reasoning they demand, and so on – are vast and still being explored.

Therefore, a programming language is less a purely mathematical object and more like an object found in nature. In addition to any formal interfaces it may present, we should – and do – also view it as a target for experimentation. We subject it to a variety of tests. Most of all, we follow a loose version of the scientific method: we form a hypothesis about the language's behavior; we construct tests (example programs and their expected outputs) consistent with the hypothesis; if the tests pass, we reinforce the hypothesis, otherwise we find it falsified; we use the falsification to generate a new hypothesis, which results in new tests. When we have exhausted our ability (or energy) to falsify, we accept our hypothesis as a tentative truth about the language, and use it to construct actual programs (whose behavior may – after painful debugging sessions – again falsify our understanding).

4 Mystery Languages for Teaching

To bring this mindset into *teaching* programming languages, we have started to experiment with what we call *mystery languages*. (This perhaps unfortunately abbreviates to "ML", though the allusion to machine learning is not unintentional; better suggestions welcome.) A mystery "language" is one syntax with multiple semantics that explore the design space for a feature. The student's task is to tell apart the differences and categorize them.

4.1 What They Are

Each mystery language exercise follows this template:

- A specification of a syntax. Each syntactic term is accompanied by an intentionally vague textual description of its semantics. (The reason for intentional vagueness will soon be made clear.)
- An editor for writing programs in that syntax.
- A parser for that syntax.
- Implementations of *multiple different behaviors for the same syntax* – we call these the *variants*. Through experiments, students will have to figure out how these differ.
- A display area to show the output of each variant.

There are two reasons for vagueness. First, the description must encompass all the variants. Second, it mirrors the vagueness sometimes found in real-world language documentation, where the meaning is clear to the author but can be misleading to someone coming from a different mental model of the same feature.

As a simple example, suppose a language introduces a notation like `a[i]`, and the description says that this is an array dereference. When `i` is within bounds, all variants might behave identically. But when `i` is out of bounds, one variant might signal an error (like Java does), another might return a special undefined value (like JavaScript does), and the third might return the content of some other array (like C sometimes does).

In practice, the mystery language variants are much weightier than this, and primarily concentrate on "normal" rather than error behavior. For instance, they present different semantics for argument evaluation, for structure mutation, for inheritance, and so on. Space precludes us from discussing them in detail, but readers may find it instructive to see (and even play with!) several concrete examples. Look for the "ML:" assignments on:

`http://cs.brown.edu/courses/cs173/2016/assignments.html`

This idea also *scales down*: in the very first week, with just numerals and a few binary operations, students can already explore the different semantics given to numbers (all floating point à la JavaScript, a mix of integer and floating point as found in most mainstream languages, bignums and rationals à la Scheme – as well as behaviors for division by zero, ranging from an error to a value representing infinity to a value representing undefinedness).

Certainly, discriminating variants is not new. For decades, texts have asked for or shown pithy examples that illustrate, for instance, the difference between call-by-name and call-by-value. What we have done is taken this idea to the extreme to explore its consequences: We examine dozens of variants; we actually implement them, so they don't just live on paper; and we put them in a consistent syntactic (Section 4.2) and execution framework so that students can focus on just what is changing (the semantic variants) without the distraction of changing syntax, execution environment, etc. These quantitative changes add up to a qualitative difference, enabling a new kind of pedagogy.

4.2 Assignment Prompts

Because a language has many parts, the assignment statement directs students to the parts on which they should focus their exploration. We have chosen – though this is by no means necessary – to grow a single language (syntax) incrementally, so each new assignment introduces some new syntactic features. Naturally, the focus is primarily on that new feature. However, many new features interact with any or all the other (previous) features of the language, so in studying that new feature, the student must explore its interactions too.

Given a set of variants of a mystery language, a student's task is to somehow figure out how they differ – this is where they engage in *adversarial* behavior – and then explain these differences (the *science*). Students must thus submit their solutions in two parts:

Classifier. A small set (usually no more than five) of programs, called *discriminators*, in the common syntax. Each discriminator must produce different output on at least two variants, and between the entire classifier, all variants must be distinguished.

Theory. A prose explanation of what they think the underlying behavior is. This is where the scientific method kicks in: they must formulate a theory and defend it using their classifier. Therefore, a good solution does not just turn in the first classifier found, but rather goes through several classification-theorizing (i.e., concrete-abstract) iterations until, ultimately, the examples are able to support the provided theory.

It can be tempting to produce the smallest classifier possible – e.g., by combining several discriminators into one. The theory acts as a counterweight against this. To provide a clear description, it makes sense to keep different explanatory programs separate.

4.3 Rules for Language Variations

Because students are being given black-box implementations with essentially undefined behavior, they have no way of knowing how complex or perverse a language variant might be. We therefore adopt the following rule. Every variant is tagged as one of the following:

Core. A Core variant has behavior (for the feature of interest) that was or is found in some widely-used, mainstream language. This does not mean the behavior could not be considered perverse; it simply means that at some point in time, many programmers were exposed to it. (This can, for instance, include dynamic scoping, or functions without proper recursion.)

Advanced. Advanced variants are similar to Core ones but are, in our judgment, either uncommon or especially difficult to find. For instance, an Advanced variant might mimic

the vagaries of "lifting" variable declarations in scope in JavaScript, or might implement call-by-copy-result on function calls.

Prank. Prank variants are where we have fun. They might, for instance, turn some identifiers into Roman numerals, introduce laziness into some positions (in a language with mutation), change case-sensitivity, or alter evaluation order. (Sometimes, real languages have subtle implementation errors that result in bizarre behavior; these would also be good candidates for pranks.)

Importantly, we inform students that they: are expected to classify all Cores; should try to classify as many Advanced variants as they can; but, to get a good grade, are *not* expected to classify *any* Prank variants. This way, students with time or motivation can explore the Pranks, but most students can avoid them and prevent the assignments from turning into frustrating and bottomless time-sinks exploring arbitrary perversity.

4.4 Linking to Lectures

There are several ways in which the mystery language homeworks can interact with the lecture schedule. One approach we found especially productive was to have one or more lectures on each mystery language assignment immediately after it was turned in, while their work (and struggles) were fresh in students' heads. The lecture consists of:

- Live classification using discriminators provided by students in the class.
- A revision of these programs into a more "canonical" classifier, if necessary.
- A description of the expected underlying theory, with ties to the classifier.
- A discussion of the broader perspective surrounding this theory: one part historical (which languages did or do this) and one part design (why they did or do so). It is valuable to present these as non-judgmentally as possible: they all had advocates with good reasons at some point. Reconstructing their thinking, and showing why it is no longer relevant, is often much more instructive than simply dismissing it out of hand. (Features like dynamic scope, or COBOL's approach to "recursion", or multiple inheritance, might all fall in this camp.) In particular, one hopes this will help students recognize these arguments if they or their colleagues make them in a language that they subsequently design!

The lecture is also the ideal time to introduce the relevant vocabulary: e.g., "static scope", "aliasing", etc. These terms, which may have seemed rather abstract and perhaps irrelevant beforehand, now have an urgent value, because they precisely capture the concepts students both observed in their discriminators and struggled to describe in their theories.

4.5 Grading Criteria

One part of the grading is trivial: telling whether the classifier properly classifies. Indeed, the beauty of this part is that it is essentially self-grading: at the time of submission, students already know how well they did (and can seek help from course staff right away).

Grading the theory, of course, requires knowledge and judgment. The graders need to know what the "true" differences are, but should also be aware that there may be more than one accurate way of describing those differences. They then assess how well a student's theory predicts the variants' actual behavior, and how well articulated it is. Of course, when all discussion of the feature is deferred to after the assignment is due, students fundamentally lack the vocabulary to describe what they see. Grading then measures how well they were able to articulate the *idea* behind such concepts even though they lacked a crisp term for it. This is not easy to grade, but ease of grading should not always be the primary criterion.

5 Experience

We have now experimented with variations of this approach in two courses at two different universities, with preliminary success. At WPI it was used as a small component of the course; at Brown, it was the primary structure of the whole course. Here, we report on data from Brown. We believe a detailed statistical analysis would suggest false precision; instead, we offer a broad-strokes summary.

The course at Brown is primarily taken by juniors and seniors (3rd and 4th year college students), but also some sophomores (2nd year), master's students, and PhD students. The students have very diverse backgrounds: some have seen primarily Java and some Python; some have seen Racket, OCaml, and Java; some have seen Pyret; many have seen low-level programming in C; and so on. This class had about 70 students turn in each assignment (though some were done in pairs).

On almost every assignment, all but 2–3 students successfully classified all the languages. The exceptions were: missing parameters being treated as undefined values (à la JavaScript); COBOL-style non-re-entrant function calls; call-by-copy-result; shallow-copy on calls for structures; and mixing laziness with state. For these, about 10% of the class failed to fully classify the languages (but still distinguished most variants). An assignment with several different semantics of field access (corresponding to Python, JavaScript, R, etc.) saw the biggest variation, but this is because only two were tagged Core, one was Advanced, and the other two Prank. (Still, $\frac{1}{3}$ of the class classified them all.)

In terms of their theories, until about half-way through, as many as half the students got grades that indicated notable weaknesses (but better than nothing). As students improved, we altered our grading scale to provide more refined information. In the second half, about 60% obtained an A grade, 35% obtained a B, and a handful got C's and A+'s (superlative).

How long did the work take? About 95% spent under five hours on each assignment (self-reported), with half or more (depending on the assignment) spending under two hours. (For simpler homeworks about a quarter reported spending under an hour, but these durations vanished on the more difficult assignments.) Every assignment had about 5% spend 5–10 hours, and virtually nobody ever reported spending over ten hours.

Students were also surveyed at the end of the semester. They reported that the mystery language assignments:

- Gave them tools to confront a new language:
  50% strongly positive, 42% somewhat positive
- Helped them separate syntax from behavior:
  62% strongly positive, 20% somewhat positive
- Helped them learn about possible behaviors of the features they studied:
  62% strongly positive, 29% somewhat positive
- Helped them learn to judge between different behaviors:
  48% strongly positive, 48% somewhat positive

In short, there were very few negative sentiments, and the assignments met the course's learning goals overwhelmingly well.

Students were also asked whether the assignments were frustrating and whether they were fun. Thankfully, students did not view these as contradictory. Approximately: 19% found them very frustrating, 48% somewhat frustrating, 29% not very frustrating, 1% not frustrating at all. Simultaneously: 29% found them very much fun, 48% somewhat fun, 14% not much fun, and 1% not fun at all. (That one student intensely disliked almost all aspects of the course, across the board.)

6 Perspective

Having discussed the specifics of these assignments in considerable detail, it is now worth revisiting the claims made in the abstract and introduction and seeing how well the assignments measure up.

First, we should justify the paper's title. Our approach is clearly experiment-centric, and we believe that it also has notable parallels with the view taken by people who break into systems. The mindset that suggests combining disparate and dissociated elements to explore their outcomes is essentially at the heart of mystery languages, too.

What our results (Section 5) suggest is that the vast majority of students who opted to take this new format of course are able to engage in this behavior. These are not students who were selected for the "security mindset" (most had not taken the hacking-oriented security course; the course had similar enrollments and drop rates as previous years; etc.). There were also no discernible biases in outcomes. For instance, women did not perform worse than men; proportionally, they actually did better. (This cannot be explained by grader bias, because all grading was anonymized.) Furthermore, the majority of the class found it constructive to engage in this activity, and even found fun in the frustration.

Student performance and the survey results show that this approach was very successful at helping students separate syntax from semantics, enabled them to explore in some depth the features they confronted, and gave them a framework for exploring language designs.

As far as exploring new languages, the final course project asked them to choose the "object-oriented" features of one of JavaScript, Ruby, Python, or R, and (with no scaffolding from us) write up a descriptive report accompanied by illustrative examples "highlighting any behavior that seems unconventional or peculiar", and discussing their consequences for type system design. This work was graded by a team mostly excluding this paper's authors. Students averaged 80% with a standard deviation of 22% – a reasonable outcome considering that this was the first time all semester they were asked to perform such a task.

In addition, mystery languages appear to have the following virtues:

Modularity. Mystery languages can be used with minimal curricular dependencies. The WPI course was and remained mostly implementation-based, but was able to incorporate a small quantity of (a variant of) mystery languages. The Brown course (which had previously been entirely implementation-based) switched mostly to this, but also had a few implementation assignments in parallel. Mystery languages can also be injected into a paradigms or other style of class, being used to concretely illustrate certain points.

Incrementality. As the WPI experience shows, one does not need to change the structure of their course entirely. A few select features can be explored using mystery languages, after which the course can either grow their use or not. In particular, instructors can give them a try without having to change their classes wholesale. Furthermore, our data show that a few hours are sufficient to complete a mystery language assignment, making the base footprint very light.

Interoperability. Mystery languages interface well with other approaches. In a paradigms approach, for instance, they can be used to make certain salient issues concrete, and to let students explore them through programs rather than just on paper. In an interpreter approach, they can be used to introduce and make real a feature before students write a definitional interpreter for it – and also give them a ground-truth against which to test their interpreter.

Complementarity. Mystery languages introduce a new way of thinking about languages. These don't displace talking about paradigms or exploring the fine structure of languages

through individual expressions in an interpreter, but rather complement it. While in principle similar points could be made through, say, writing different interpreters for one syntax, the burden of building several languages feels much greater than that of writing a few test programs – and, to many students, may also be less fun, since it involves more "building" and less "breaking".

Therefore, we believe it would be profitable for other instructors to also consider incorporating this approach. Indeed, we would ideally hope to build a large communal repository of mystery languages so that instructors can pick-and-choose based on taste, interests, and also the need to avoid plagiarism.[1]

7 Challenges

That's the good news. There are also many challenges induced by this approach that we need to explore over time.

- None of our claims have controls. Perhaps students would meet the same learning objectives just as effectively through other teaching models. Perhaps we are naturally biased in the selection of questions and of metrics. Perhaps this approach does not work at other universities. We feel a larger community evaluation effort is needed to obtain definitive answers to some of these questions.
- It is tremendously difficult to build and maintain all these languages. This semester alone, we essentially built 36 different ones![2] Furthermore, we built them in JavaScript for student convenience (and you, too, can try them – right now – from the URL given above, as did many course alumni). Unfortunately, JavaScript is hardly a natural medium for building programming language implementations, much less variations between them. A language laboratory like Racket [3] would have been a much better choice in principle. However, the choice of language must be balanced against deployment issues – especially early in the life cycle of this approach, when errors are sure to creep in and will necessitate rapid and painless re-deployment of fixes.
- It can be difficult for students to formulate a theory about black-box languages, and to express it without suitable terminology at their command. (In particular, we do not yet have good means to help students who are utterly stuck.) It would have been constructive for them to somehow redo their work after the accompanying lectures.
- It is difficult to have one syntax apply across all variants. This can lead to slightly perverse choices: e.g., for field access, we used the syntax `o["x"]` for all languages, even those where the field name is first-order – where `o.x` would have been much more natural. (The first-order languages imposed syntactic restrictions on what can appear inside brackets, which otherwise appears to be an expression position.)
- We have had time to explore only a small space of language features. Many more, such as types, concurrently, and advanced control, remain untouched.

Nevertheless, we view these as challenges, not as insurmountable obstacles.

Acknowledgements. We are deeply grateful to Anjali Pal, Sahakait Benyasut, Sara Hartse, William Kochanski, and Alex St. Laurent, the Brown course undergraduate TA staff, for

[1] The danger of plagiarism is in part why we do not describe the languages in more detail in this paper: there are only so many non-prank variants one can create, after all!

[2] Some feature interactions due to implementation shortcuts unfortunately resulted in stranger language semantics than even we expected – and which students caught...

enthusiastically taking the plunge with this new approach. We especially thank William for helping us formulate the written portion of submissions in terms of the scientific method, which provided much-needed guidance and clarity. We also thank Eli Rosenthal for a useful discussion that helped trigger this approach. We are grateful to Matthias Felleisen, Eli Barzilay, and Preston Tunnell Wilson for useful conversations and feedback.

References

1 Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.
2 Gregory H. Cooper, Arjun Guha, Shriram Krishnamurthi, Jay McCarthy, and Robert Bruce Findler. Teaching garbage collection without implementing compilers or interpreters. In *ACM Technical Symposium on Computer Science Education*, 2013.
3 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket manifesto. In *Summit on Advances in Programming Languages*, 2015.
4 Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press.
5 Samuel Kamin. *Programming Languages: An Interpreter-Based Approach*. Addison-Wesley.
6 Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*.
7 Shriram Krishnamurthi. Teaching programming languages in a post-Linnaean age. In *SIGPLAN Workshop on Undergraduate Programming Language Curricula*, 2008. Position paper.
8 John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(3), 1960.
9 Robert Sebesta. *Concepts of Programming Languages*. Addison-Wesley, fifth edition.