Centering Humans in the Programming Languages Classroom: Building a Text for the Next Generation

Rose Bohrer
rbohrer@wpi.edu
Worcester Polytechnic Institute
Worcester, Massachusetts, USA

Abstract

This paper is a companion to the author's open-access textbook, "Human-Centered Programming Languages." Beyond the contributions of the textbook itself, this paper contributes a set of textbook design principles for overcoming those limitations and an analysis of students' stated needs and preferences drawn from anonymous course report data for three courses, the last of which was based on notes that became the basis of the textbook. The textbook is intended to be multi-purpose, with its primary audiences being undergraduate and master's-level elective courses on programming languages within computer science, but significant opportunity for cross-use in disciplines ranging from human-computer interaction and software engineering to gender studies and disability studies. The book is intended to be language-agnostic, but the course in which it will be used first is Rust-based.

CCS Concepts: • **Applied computing** → **Education**; *Media arts*; *Fine arts*; *Arts and humanities*; • **Theory of computation** → Formal languages and automata theory; Type theory; Program semantics.

Keywords: programming languages education, textbook design, interdisciplinary curriculum, human-centered computing

ACM Reference Format:
Rose Bohrer. 2023. Centering Humans in the Programming Languages Classroom: Building a Text for the Next Generation. In *Proceedings of 2023 ACM SIGPLAN International SPLASH-E Symposium (SPLASH-E).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SPLASH-E, October 25, 2023, Cascais, Portugal
© 2023 Association for Computing Machinery.
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Who takes a course on programming languages (PL)? For decades, this question has driven the design of PL courses and the textbooks that support them. This question cannot have a single correct answer, but every instructor must answer it to the best of their ability. For the sake of rhetoric, consider three potential answers:

1. PL courses are taken by students who want to learn many different languages.
2. PL courses are taken by students who want to prepare for research in programming language theory.
3. PL courses are taken by students who want to become programmers who understand their tools.

This paper proposes a radical fourth definition of the audience: *PL courses are taken by students who will work in interdisciplinary teams, and who must work across disciplinary boundaries.* These boundaries often lie within computing: my recent offering of graduate PL was only 25% CS grad students, with many students from Data Science, Computational Media, and Robotics. At WPI, Human-Computer Interaction (HCI) is not separated from CS, thus HCI students may be present as well. Yet doing justice to this breadth requires looking outside computing to social sciences and humanities as well: Interdisciplinary programs build as heavily on these disciplines as they do computing. HCI relies heavily on the social sciences to comprehend the human experience of computation. Computational Media relies heavily on the humanities, e.g., media studies, rhetoric, and aesthetics.

If we wish not to triple the workload of a course, this shift requires radical restructuring of that course. Beloved topics from PL theory will be dropped to make way for breadth. In this new approach, the core criterion is *literacy:* we wish to prepare students to read work from multiple disciplines. If we have equipped students for self-study, the removal of specialized topics becomes more acceptable. To encourage self-study without overworking students, the course blends teacher-regulated and student-regulated learning.

This paper lays out the construction of a *human-centered programming languages* (HCPL) textbook, structured around the variety of humans who care about PL, primarily structured through the variety of disciplines and roles for each person. This contrasts with the complementary notion of *user-centered* programming languages [6, 7, 21, 30, 31]: in

HCPL, users are but one group of humans who are put into conversation with others.

The HCPL textbook [3] is an open-access living document, released at the same time as this paper. Its preliminary form was ≈50,000 words of course notes for grad PL in Spring 2023; since then it has grown to ≈100,000 words and is suitable for production use, though it will continue to evolve. The book is available free-of-charge at https://bookish.press/hcpl. Related course materials are openly available at https://github.com/rbohrer/pl-course.

The HCPL approach is fundamentally interdisciplinary. As a result, this approach also constitutes a reconsideration of what a PL course can be. PL Education in the past has made great strides in the development of PL *depth* courses and textbooks [17, 28, 35, 36, 38] which provide specialized technical knowledge. Though those courses will not go away, HCPL breaks new ground by fundamentally reimagining a PL course as a *breadth* course, which succeeds not when it creates specialists but when it helps students identify their interests within a range of options and work effectively across differences of interest. This is *scholarship of integration*. When HCPL places PL theory, HCI, and critical theory in the same textbook, it is not innovating in any one of these topics, but innovating in bringing them together.

The following subsections of the introduction identify pedagogical needs at the author's institution, WPI (Section 1.1), identify formal learning objectives (Section 1.2), and provide a positionality statement (Section 1.3). We highlight core pedagogical strategies used in the textbook in Section 2, analyze course report data in Section 3, compare this work to the history of PL education and related work in Section 4, discuss how the book might be applied in the classroom in Section 5, and conclude in Section 6. The course report data are drawn from three of the author's courses Introduction to Program Design, Undergraduate Programming Language Design, and Graduate Programming Language Design.

1.1 Local Needs

The population of computing students at WPI informed the design of *Human-Centered Programming Languages*. We list relevant traits:

- Undergraduate courses at WPI run on 7-week terms (named A,B,C, and D) and graduate courses operate on semesters. The book must provide flexibility in the amount of material covered and the speed of coverage.
- Prerequisites are soft. Courses may have students who have not yet studied recommended material.
- Most students have little proof experience, but some will desire it.
- Students are historically strong at programming, with an eagerness for project-based work. Many of these students proceed to become software developers.
- Our master's programs traditionally admit students from a wide range of undergraduate majors and must support students who are undergoing a transition in their topic of study
- We have few PhD-specific courses, yet we have a significant population of PhD students, and they take master's-level courses, often across disciplines. Thus our courses must accommodate specialists in training at the same time as transitioning students.
- Our computer science department's gender diversity is well below parity (25% non-men) [11] and racial diversity is low (11% URM) [11]. Approximately 19% of students have disability accommodations [11]. Statistics on LGBT+ students are unavailable, but lived experience suggests a substantial population. This combination of factors means a substantial body of students may be marginalized and may benefit from explicit gestures of inclusion.

The redefinition of PL education as fundamentally interdisciplinary is motivated by this concrete pedagogical context, with the hope of generalizing elsewhere.

1.2 Learning Objectives

The following learning objectives were identified in the syllabus of my graduate PL course in Spring 2023, which was taught using notes that became the basis of the textbook:

1. Identify problems where programming language design can be used
2. Communicate with clarity and technical depth about language design
3. Develop a mathematically-precise definition of a language's syntax
4. Develop a mathematically-precise definition of a language's semantics
5. Implement interpreters for programming languages
6. Situate your own work among the schools of thought [archetypes] discussed in class

and the following classwork was outlined in the syllabus in support of those learning objectives:

- Implement parts of a programming language (parsing, evaluation, type-checking)
- Perform miniature language design exercises
- Perform a usability study about language design with your classmates, and
- Engage critically with academic research about the intersections of human-computer-interaction with social issues as it pertains to programming languages

These learning objectives reflect the multiple nature of the course: some students will value implementation, some will value theory, and all will need to operate in an indisciplinary setting. Though the textbook is designed with these learning objectives and classwork in mind, it does

not outright eliminate the fundamental complexity of teaching in an interdisciplinary setting, which must be supported through course design. Teaching grad PL in Spring 2023, an ungraded [13, 22, 41], project-based [23] structure was used to accommodate this interdisciplinary setting, but teaching methodologies are expected to evolve over time.

1.3 Positionality Statement.

The author's identity informs the work. When HCPL addresses disability, it speaks from her experience of Ehlers-Danlos Syndrome and neurodivergence. When HCPL addresses gender, it speaks from her experience as a trans woman. At the same time, best efforts are made to include other marginalized people, without speaking over them:

- the treatment of gender and disability builds on intersectional perspectives regarding race and culture, as well as non-binary perspectives,
- open access removes economic barriers, and
- the author specifically sought out feedback from programmers from countries underrepresented by computer science academia (e.g. Indonesia).

This is not a claim to perfection. The author wrote the book because she wishes for it to exist, and any actually-existing book will need ongoing revision for inclusivity.

2 Pedagogical Strategies

We describe the strategies taken in the construction of the textbook to support its learning objectives, such as the use of archetypes as characters who represent different humans that study programming languages (Section 2.1), the choice of topics (Section 2.2), the use of dialogues between the archetypes (Section 2.3), and the design of exercises (Section 2.4) for applicability across multiple disciplines.

2.1 Archetypes

In contrast to user personas [33], the five *archetypes* of HCPL represent multiple ways of engaging with PL through different roles and academic disciplines. These archetypes are not meant to be rigid or exclusionary: instead of forcing students to "pick a side," the goal is to help students locate their own perspectives among a menu, locate their classmates' perspectives, and communicate effectively even when perspectives differ.

The five archetypes are the Practitioner, Implementer, Theorist, Social Scientist, and Humanist. The textbook introduces each archetype by discussing their values and what questions they seek to answer. We do the same here.

The Practitioner: The Practitioner is someone who interacts with code as a programmer, but does not implement, design, nor theorize about programming languages. It is essential that the Practitioner not be stigmatized nor viewed as lesser than the other archetypes. A central point of the book is that many in the audience fall into this group, that they experience programming languages in a distinct way from the other archetypes, and that this valid experience can be put into conversation with the others.

The book provides pure speculation of potential Practitioner traits; grounding these traits in evidence is ongoing work. We cite lived experience as a primary way that the Practitioner forms beliefs about programming languages: they experience code by writing and reading it, and their joys or frustrations in this process shape their beliefs about languages. Moreover, the Practitioner is often goal-directed: They wish to solve some problem by implementing software, and their interest in programming languages is primarily as a means to achieve this end.

The Practitioner's fundamental question is "How do I write this program?"

The Implementer: An Implementer is anyone who implements a programming language, typically as a compiler or interpreter. By definition, they are simultaneously a Practitioner, but they have an additional set of needs and concerns. The Implementer is heavily concerned with tools for building languages. The implementer might use regular expressions, context-free grammars (CFGs), and parsing expression grammars (PEGs) to implement a parser and abstract syntax trees to represent parsed programs. The Implementer needs a strong understanding of a program's meaning (semantics) insofar as they need to faithfully capture a program's meaning in the implementation.

The Implementer's fundamental question is "How do I implement this programming language?"

The Theorist: The Theorist says that PLs are formal languages that can be defined and analyzed mathematically. A "good PL" is a language that we can analyze in powerful ways. A Type Theorist is a Theorist who believes a "good PL" has a rich static type system that lets us prove powerful theorems about the correctness of programs. The Theorist understands programming language design as the task of abstraction-building, the discussion of programming languages as abstraction-criticizing, and programming as the use of abstractions.

The Theorist's fundamental question is "What can I prove about this language?"

The Social Scientist: The Social Scientist's emphasis is on the rigorous academic study of humans. The Social Scientist may study humans for humans' sakes or for the sake of informing technical solutions to the users' problems. They could be embedded in computing fields such as HCI and Software Engineering, or they could belong to standalone fields such as Linguistics and Psychology. They often study social issues within communities of programmers and computer users. Who do these communities include or exclude? Why? What could be done about that? These questions are answered using scientific approaches, which can encompass both quantitative or qualitative approaches. The Social Scientist's fundamental question about programming languages

is “How do programming languages affect communities of people?”

The Humanist: The Humanist also studies society. For this reason, some computer scientists ask how HCPL distinguishes the Humanist from the Social Scientist. In these HCPL archetypes, the most important differences are methodology and motivation. Our Social Scientist draws heavily on qualitative methodologies like surveys, interviews, and observation of subjects. Our Humanist relies primarily on critique, which often involves analyzing code as a written text. Our Social Scientist is often situated within HCI or design settings, with a goal of using their analysis to directly inform development of new artifacts. Our Humanist is often situated outside these settings, with critique as its own ends. In the context of HCPL specifically, the Humanist may be situated within Gender Studies or Disability Studies. As with all the archetypes, real-world Humanists are more varied than a single character can portray.

To clearly distinguish the Humanist from the Social Scientist, we could frame the Humanist’s fundamental question as: "How can social critiques be applied to PLs?"

These five archetypes are chosen with intention. The inclusion of the Social Scientist and Humanist makes the work firmly interdisciplinary. The resulting scope is wider than the related concept of *User-Centered Programming Languages*, an approach that uses the methods of the Social Scientist to tend to the needs of the Practitioner. Instead, our emphasis is on putting these archetypes into conversation with one another. While these archetypes are interdisciplinary, their disciplines are not equally distributed: three archetypes lie solidly within Computer Science (the Practitioner, the Implementer, and the Theorist). This is also intentional: the book focuses on the needs of CS courses which wish to increase their breadth, and thus CS has an outsized role.

The archetypes are used in two main ways: in chapter outlines and summaries to help contextualize each chapter within the broader aims of the book, and in literal dialogs between the characters intended to engage the reader directly in interdisciplinary discourse (Section 2.3). For an example use of the archetypes in a chapter outline, consider an outline for the Regular Expressions chapter:

- The Implementer cares about regular expressions because they are a core tool for implementing a programming language, specifically the part that parses basic building blocks of syntax
- The Practitioner cares about regular expressions because they are also used in a wide variety of small text-processing tasks in programming, such as parsing email addresses, IP addresses, dates and times, passwords, plaintext data files, and more. They can also be used to efficiently search through large bodies of text, such as the code you write.
- The Theorist cares about regular expressions because their simplicity is their strength. Regular expressions are simple enough that many properties about them can be computed by programs, making it possible to design algorithms and tools that make the Implementer’s and Practitioner’s jobs easier.
- The Social Scientist might care which uses of regular expressions in programming are most common, and whether they are effective.
- The Humanist might care about the communicative limits of regular languages as tests, and how the assumption of a regular language places structural limitations on ability to communicate.

The goal of the outlines it not to be exhaustive nor prescriptive, but to expose students to multiple motivations and multiple truths which connect with different readers.

No.	Title	Length
1	Introduction	10min
2	What is a Language	25min
3	Programming in Rust	1hr
4	Regular Expressions	35min
5	Context-Free Grammars	50min
6	Parsing Expression Grammars	30min
7	Abstract Syntax Trees and Interpreters	35min
8	Operational Semantics	35min
9	Types	50min
10	Users and Designers	40min
11	Quantitative Methods & Surveys	45min
12	Qualitative Studies	15min
13	Gender	30min
14	Disability	20min
15	Media Programming	15min
16	Play	15min
17	Natural Language	25min
18	Diagramming	15min
19	Process Calculus	20min
20	Cost Semantics	20min

Table 1. Table of contents for current iteration of book

2.2 Topic Selection

We list the chapter titles and length estimates[1] in Table 1 on page 4, group them into sections by theme, explore how they were chosen, and suggest how they could be used. The chapters can be grouped in the following way:

[1]The length estimates are provided automatically by the publishing platform, Bookish.press. Reading time varies by reader, but these estimates give a sense of relative length.

- Chapters 1–2 introduce the book and its approach to programming languages.
- Chapter 3 is a Rust [20] primer. It prioritizes the Practitioner. It is the only chapter that focuses extensively on a particular language; later chapters are predominantly language-agnostic with the language-specific details separated out from the main text. Primers for other languages could be added later.
- Chapters 4–7 prioritize the Implementer. By the end of these chapters, students can implement a parser and interpreter for a basic impure functional programming language, which is Turing-complete.
- Chapters 8–9 prioritize the Theorist. By the end of these chapters, students can translate between implementation and formal inference rules, including the implementation of a type-checker when given rules.
- Chapters 10–12 prioritize the Social Scientist, particularly through the lens of HCI. By the end of these chapters, students can design and execute a short user study to address a research question about programming languages.
- Chapters 13–14 prioritize the Humanist, drawing heavily on perspectives from gender studies and disability studies. By the end of these chapters, students can describe how major concepts from gender studies and disability studies can influence discourse about programming language design.
- Chapters 15–20 are breadth material. The first four are case studies of specific issues in design through specific projects such as Processing [37], Twine [14], Flow-Matic [42], Inform [32], and Penrose [44]. The last two chapters are breadth topics in the theory of programming languages.

To choose which material to *include*, we require 1) that every archetype get the leading role in at least one chapter, preferably two, and we require that 2) the book should meet the learning objectives. These criteria suffice to justify chapters 1–14, the heart of the book. The remaining chapters were chosen through the requirement that 3) breadth material should be tailored to (inter)-disciplinary students' interests. Chapters 15–18 aim to include Computational Media students and chapters 19–20 are an invitation to further PL theory study.

To choose which material to *exclude*, the guiding principle was literacy over operational ability. For this reason, proofs are nearly absent from the book chapters, appearing only in an optional section of Chapter 9. We focus on teaching a student to comprehend the statement of a proof rule or theorem, not prove it. Advanced topics from each field are excluded: no dependent types, no optimizing compilers, no participatory design workshops, no code-as-creative-writing [27]. Instead we focus on the core vocabulary of theory, implementation, design, and critical studies, with ample references for self-study.

The chapter order suggests several possible course schedules. For a semester-length interdisciplinary course (e.g., PL + HCI), chapters 1–14 are recommended as the core. Supplementary material should be chosen based on instructor and student interest, drawing from Chapters 15–20 or elsewhere. Chapters can be used in disciplinary courses: CS (1–9), HCI (10–12), gender studies (13), and disability studies (14).

2.3 Dialogues

Dialogue is an ancient teaching technique, since at least Socrates. The Socratic method [1, 9] relies on the use of questions to encourage a student to arrive at a correct answer. However, Socratic dialogues assume a fixed truth which can be predetermined by an instructor who then merely helps the student arrive at the fixed truth. Dialogue has even appeared in PL-adjacent curricula, e.g., through *Proofs and Refutations* [26].

The dialogues in HCPL do not seek a fixed truth. Each dialogue starts with a question that might serve as a classroom discussion topic, such as the value of properties like speed or safety, or the value of formalisms like regular expression, context-free grammars, and parsing expression grammars. Instead of constructing one true answer to each question, however, these dialogues serve to deconstruct a question and reveal the variety of truths which hold for each of the archetypes. These dialogues typically resolve with a greater shared understanding, but not necessarily agreement. As an example, we present a dialogue from the chapter on Context Free Grammars, which starts with the question "Do context-free grammars model human language?" Statements by each archetype are annotated with their initials (P: Practitioner, I: Implementer, T: Theorist, SS: Social Scientist, H: Humanist).

> **Q:** Do context-free grammars model human language?
> **SS:** Of course they do. I invented context-free grammars and I invented them to model natural human language. That's what's so amazing about them, they can model every natural language that has ever existed.
> **T:** Woah! Did I just hear you make a universal claim, SS? I never thought I'd see this day. Can I prove this by induction?
> **SS:** I won't stop you from trying, but that's not how I went about it. We have data from lots of different languages, and we've found the grammar for all of them can be expressed as context-free grammars. Speakers of those languages all agree with us.
> **H:** I never thought you would have me agreeing with T, but desperate times call for desperate

measures. You and I both know not all human language is context-free. What about Swiss German? [40]
T: A proof by counter-example!
SS: Fine, fine, but if it works for most languages, it has to be useful, right?
I: Useful for whom?
SS: Me? You? Anybody who speaks? Context-free grammars are really flexible, I'm surprised at all this backlash...
I: I'm not teaching a computer to talk, I'm teaching it to run code. I don't want a machine to be flexible, I want it to be precise.
P: I'm with you on this one. How else would I understand the documentation for a new programming language?
SS: Context-free grammars are precise, but they describe a whole set of possibilities. You need that kind of flexibility for natural language. I won't stop you from using other tools for programming, but you're going to need this for natural language.
P: Sorry for ganging up on you, SS. I just started writing a chatbot, and I think I'm starting to see your point.
T: Sorry... Can I still do a proof?
SS: Actually, yeah, that's the benefit of a simple formalism.

In this dialogue, every archetype had something to say; in others, as few as two might speak. Each speaker brings a valid experience to the conversation: **SS** brings linguistic expertise, **H** brings critique, **T** brings careful logical reasoning, **I** brings awareness of language implementation needs, and **P** brings awareness of practitioner needs. The dialogue does not necessarily conclude with agreement: the Implementer is not required to prefer context-free grammars, the Social Scientist is not required to give them up, and the Theorist and Humanist are not required to believe a universal claim. In this case, the dialogue provides a natural segue to the following chapter on parsing expression grammars (PEGs), which address the Implementer's concern with non-deterministic grammars. In this case, as with many real-world dialogues, the discussion eventually diverges from the initial topic. This is an intentional choice.

The success of dialogues in the classroom depends on how they are integrated. We propose two potential uses. Firstly, the dialogues can be assigned as pre-class reading to prepare students for roleplay activities in class where they debate a closely related topic from the perspective of each archetype. Secondly, handouts of student-regulated language design assignments can include the dialogues on a list of suggested readings for students seeking design inspiration.

2.4 Exercises

The design of exercises is foundational to the success of a textbook. To meet the unique interdisciplinary needs of HCPL, the book takes a mixed approach to the design of exercises. Examples are drawn from mixed domains, aiming to cover the archetypes and cover the interdisciplinary programs from which students come.

Exercises range from brief checks of understanding to full-fledged research projects, which are indicated as such. We describe several notable styles of exercises from the book, which center CS but allow incorporation across multiple disciplines:

- **Traditional exercises:** We include many traditional exercises for PL courses such as implementing short functions in a new language, defining regular expressions, CFGs, and PEGs that implement a specification, defining inference rules for new language features, and (for courses where students have prior proof experience) proving classic theorems. We wish to retain what works from traditional approaches.
- **Implementation by Translation:** Implementation is a traditional exercise, but we wish to highlight the specific structure of our implementation exercises. To support students who align with the Implementer archetype, a series of exercises are provided in which they implement a parser (including lexer), an interpreter, and a type-checker for a basic impure functional language. The algorithms for each are provided in pseudocode in the text, thus the exercise consists of translating pseudocode to the implementation language of choice. This same principle appears in the small: other exercises ask to translate inductive definitions from the text into type definitions in code, to gain familiarity with inductive definition.
- **Code Autoethnography:** To blend the interests of the Practitioner and the Social Scientist, we include structured autoethnography [10] exercises where students write about their lived experience as a programmer. They keep a structured journal of their experience with a new programming language, each time answering questions such as "What error messages, if any, confused you?" "What language features did you look up?" and "What bugs, if any, consumed the most of your time?". This journal can then be analyzed in further autoethnographic analysis. This builds on the established technique of reflective journals [15] but emphasizes the role of personal narrative and critical interpretation.
- **Critical essays:** We provide essay prompts for use in courses across the humanities. Prompts include a cultural study on programming languages that center Classical Chinese and post-structuralist comparison of static typing to either gender or genre.

- **Corpus studies:** For Social Scientists, we include corpus development and analysis exercises. Example topics include: "How are regular expressions used in production software?," and "What aspects of abstract syntax trees and intermediate representation design do Implementers focus on in online discourse?"
- **Human-Centered Theory:** We include exercises that highlight how the Theorist's methodology is compatible with human-centered values. Human-centered theory exercises include the development of provably-accessible languages, provably-ergonomic context-free grammars, and proofs of the expressive power of an interactive fiction framework (Twine [14]).
- **Design prompts:** Our exercises invite students to design artifacts such as style sheets for colorblind accessibility, questionnaires for usability questions of their choice, and static type systems for domain-specific languages in multi-media applications.

Many of these exercises are ambitious, even research-level. However, this is by design in a textbook intended for use both at the undergraduate and PhD level. The key in using these exercises is that they are written with the expectation of instructor tailoring for local conditions: in undergraduate courses, scope should be restricted and scaffolding provided. In theoretical problems, definitions and lemma statements should be provided. In design problems, key design elements can be given. In corpus studies, the instructor can suggest specific sources and an expected corpus size. For PhD students, these scaffolds can be removed in order to practice research skills.

3 Data Analysis

We carry out a qualitative analysis of written student responses in course evaluations across three courses to answer the following research questions pursuant to the development of the HCPL textbook:

1. What sentiments do students hold toward ungrading in an HCPL-based course?
2. What course delivery recommendations did students make for future iterations of an HCPL-based course?
3. What sentiments do students hold toward the centering of human issues, including social issues, in PL curricula?
4. What sentiments do students hold toward language choice in PL and PL-adjacent courses?

The following course evaluations were analyzed:

- Course A: Programming Language Design (CS 4536), A-term 2021: 14 respondents of 31 students (45%).
- Course B: Introduction to Program Design (CS 1101), A-term 2022: 46 respondents of 70 students (66%).
- Course C: Programming Language Design (CS 536), Spring semester 2023: 14 respondents of 16 students (88%).

Of these, only C used the notes that precede HCPL. Thus, these course evaluations occurred early enough to inform the direction of the HCPL project. Course A was based on *Programming Languages: Application and Interpretation* (PLAI) [24] and B was based on *How to Design Programs* (HtDP) [12], which shares an author with PLAI. Additionally, C was ungraded, relying on self-defined student projects and student self-reflection, while A and B were graded.

For this reason, the results from C receive the closest analysis. For C, we undertook a thematic analysis [4] process which consists of (1) familiarizing with the data, (2) assigning codes to the data, (3) searching for themes in the data, (4) reviewing themes, (5) defining and naming themes, and (6) writing the results. The analysis of A and B were restricted to analysis of the only questions deemed transferable to HCPL: questions about incorporating social issues in curricula and about language choice. We discuss threats to validity or limitations of the analysis, then analyze results from C, A, and B in that order.

We do not quote course evaluations out of concern that students may have assumed confidentiality. Instead, we quote from an end-of-semester email sent by one student, who explicitly agreed to publication. The email is separate from the thematic analysis.

3.1 Threats to Validity

Every study has threats to validity which limit the extent to which its results can be generalized, and the present study is no different. We highlight validity threats from sample size, response rate, and gender bias:

1. We consider 74 course evaluations from 117 students across 3 courses at the same institution. These students represent a small fraction of all PL students worldwide, and results may not generalize across all courses and contexts.
2. The response rate varies substantially across courses, from 45% to 88%. For courses with lower response rates, numerical trends in course evaluation responses may not fully represent the population. Of the three courses, the strength of the Course C evaluations is near-universal participation, but even here, the capacity to compare data between A and C is limited.
3. It is well-documented that student course evaluations are biased by perceived instructor gender [29]. This is a threat to validity because the author's perceived gender changed after teaching A and before B and C, due to gender transition. Due to this threat, we abstain from direct analysis of satisfaction ratings.

3.2 Thematic Analysis: Course C

The data source for this analysis consisted of three open-response questions which are asked on every course evaluation at WPI [emphasis in original]:

- "What did you particularly LIKE about this course?"
- "What did you particularly DISLIKE about this course?"
- "Can you suggest anything that the instructor could do to improve the quality of teaching?"

Evaluations include a fourth standard question: "Would you encourage a friend to take a course from this instructor? Why or why not?" This question was excluded from analysis because responses were disconnected from course content.

Before undertaking the thematic analysis, we prepared the data by removing responses which were fully unrelated to HCPL. This removal was conservative: if any potential relationship was identified, the data were kept. For example, data which discussed presentation aids, examples, and interactive lectures were kept, because textbooks typically influence their development. Most of the removed data were flat assertions of whether students liked the instructor, without further comments. The remaining data were interpreted through a six-step thematic analysis process guided by the research questions. This analysis was "big-Q qualitative," [5] meaning that reproducibility was not a priority. The data were coded by one person, the author, and this analysis embraces the author's subjectivity in identifying themes on which to base discussion.

We name and define the themes that were identified in this analysis. In parentheses we indicate how many responses were coded with each theme. These counts do not sum to the total number of responses, because each open-ended question was optional and we only coded responses that pertain to the research questions.

Ungrading improved learning outcomes (4). Students identified self-direction in the choice of course projects as a strength of the ungraded approach. Multiple students identified this agency as increasing their intrinsic motivation by connecting with motivating topics. In some cases, the students expressed surprise that their interest and learning exceeded expectations, with one student concluding that this approach helped them find their place within the topic. This theme was reinforced by an email from a student, we quote:

> I probably lost a lot more sleep than I should have doing some work on the project, but I only did it because I really enjoyed the freedom to design whatever we wanted. . . . Coming into this class I didn't have any prior interest or experience in theory behind languages but this definitely stoked my desire to work with it.

Though the HCPL book is not designed with the intention of students losing sleep, we wish to emphasize that this student linked agency and motivation explicitly.

Self-directed learning is in tension with desire for structure (3). Students expressed desire for structure and accountability within project-based work. Suggestions included the use of detailed timelines with specific deadlines and goal-setting processes. Though the instructor encouraged the use of these techniques, they were not enforced. These responses can be understood as a call to enforce them.

Ungrading reduced overall stress (2). Students identified ungrading as a means to reduce stress, particularly by giving students agency over the amount of time they wish to invest in a particular course. Revisiting the previous end-of-semester email, some students who experienced stress interpreted it as positive stress:

> . . . it has been amazing overall and taught me so much, especially because it was a tough term personally. What a great experience that I wanted to spend all of my time on.

Though this student may have experienced work stress, they cited the holistic experience as hoping them cope with a stressful time in life, due to their high motivation.

Ungrading is incompatible with rigorous mathematical courses (1). One student directly opposed the use of ungrading, citing it as incapable of providing intellectual challenge and incompatible with use in logic and math-related courses.

Preparation of slides, examples, and interactivity support learning (5). Students identified presentation aspects such as use of slides, examples, and interactions with students as the primary point of improvement. For context, course C was taught using paper notes and a whiteboard, but without slides, in contrast to A. This was done primarily due to time constraints, with note development taking priority over slide development. Though this theme identified a limitation of the latest course offering, it is consistent with the approach of the HCPL textbook: open publication of course materials can assist future instructors who are in need of these aids.

Interpersonal interaction was valued (2). Students reacted positively to two kinds of interaction between students in the course: one planned and one unplanned. By design, their coursework included developing a user study and carrying it out on classmates, which was viewed positively. Without planning, a pattern emerged that students would stay after class to discuss their projects with project teammates, while the instructor walked through the room to check in on each student. This was valued as well.

Breadth was valued (2). Students identified the broad range of themes as a strength, including exposure to case studies of programming languages which they are unlikely to encounter in their daily lives.

Social issues were perceived as mattering (1). One student reacted positively to the integration of social justice issues with PL education.

The human-centered approach was perceived as divisive (3). Three students' responses were interpreted as rejections of various aspects of the human-centered approach. One student criticized the focus on user studies. One alluded to the inclusion of irrelevant material but did not make direct claims. One made a direct, emotionally-charged rejection of

social issues such as gender and disability being covered in readings or classroom discussions, explicitly describing these topics as unwelcome in a computer science classroom outside the limited domain of the department's required course on social implications of computing.

3.3 Narrow Analysis of Social Attitudes: Course A

In contrast to the Course C course evaluations, the instructor added several custom questions to the Course A evaluation. The following question was deemed relevant: "In general, how would you like to see social impacts and social issues incorporated into the teaching of CS classes in the future?" For context, these issues were covered in Course A in a substantially more limited way than C. No readings nor classroom discussions were assigned relating to social issues. Instead, several of the weekly assignments included written problems tying social issues to programming languages, such as discussions of type systems for privacy and representation of gender as an algebraic datatype.

Because the implementation varied between Course A and C, we undertook a limited analysis and coded each response by whether it supported or opposed the coverage of social issues in computer science. Of 10 respondents that answered this question, 8 were in support and 2 in opposition. This is in sharp contrast to C, where 1 was in support and 3 in opposition. Conclusive interpretations cannot be drawn from these limited data, but we propose the following potential explanations:

- Students may have preferred the delivery approach of Course A over C, addressing these issues privately and as add-ons to other work, rather than as the subject of full lectures,
- Differences in response rate and sampling noise could increase variance, or
- It cannot be ruled out that students react positively to discussion of social issues when the speaker is perceived as a cisgender man (A), but not when perceived as a transgender woman (C).

3.4 Narrow Analysis of PL Attitudes: Course B

We include Course B in this discussion because HtDP and PLAI come from a shared tradition and share a language: Racket. Students were asked to give open-ended feedback for consideration by the committee tasked with curriculum design: "A committee is currently evaluating the design of our introductory courses. What, if anything, do you want to tell them?" Though the question was open-ended, students commented on language choice to the instructor throughout the term, and the instructor asserted in class that opinions on language choice could be directed to this question. We coded the responses to this question by whether they viewed Racket positively, neutrally, negatively, or whether they did not mention it at all. Of 26 responses, 11 mentioned Racket, and all 11 mentions were negative, primarily criticizing it as disconnected from future student needs.

3.5 Discussion of Results

We summarize the above results through the research questions (RQs), then discuss the implications of each in turn.

1. Ungrading and self-direction improved intrinsic motivation and reduced stress for many, but raised concerns of extrinsic motivation and rigor for some.
2. Students recommended presentation aids and scaffolding of self-directed work.
3. Students who rejected the treatment of social issues outnumbered those who felt validated by it.
4. Students in Course B identified perceived future usage as a core motivator in language choice.

In RQ1, a potential response to requests for rigor and extrinsic motivation would be to mix graded implementation work with ungraded design work.

In RQ2, the recommendations are clear: presentation aids and additional scaffolding can both be implemented in future iterations.

In RQ3, the responses could generously be interpreted as a call to explore more varied methods for the inclusion of social issues. Less generously, they reflect the fundamental nature of marginalization. Work which prioritizes a power-minority invites targeting by the power-majority. The HCPL book is unapologetic in doing so.

In RQ4, perception matters. No language is guaranteed to be viewed as a motivator. The author has argued elsewhere [2] for the motivational potential of Rust, which motivates its use in the Fall 2023 offerings of CS 4536 and CS 536.

4 Related Work

This work is developed in conversation with related areas of education scholarship: self-regulated learning, open education, and PL education.

4.1 Self-Regulated Learning

Self-regulated learning research [45] recognizes the goal of education as preparing students for life-long learning. HCPL fully endorses this goal. However, the same research recognizes fundamental challenges of self-regulated classrooms: self-regulation is a trait which can take a long time to emerge and is influenced by social context. Moreover, the removal of extrinsic motivation in the absence of intrinsic motivation can lead to decreased student satisfaction. Due to this complexity, HCPL is designed to accommodate both self-regulated and teacher-regulated learning. Exercises include self-regulatory techniques like goal-setting, strategic planning, self-recording, and self-reflection. Many technical exercises are also included which can be assessed using traditional methods. The self-regulatory exercises can be

incorporated into ungrading-based approaches, which prioritize intrinsic motivation over extrinsic motivation and criticize the power dynamics inherent to grading.

4.2 Open Education

Open education [16, 19, 39, 43] is the practice of teaching in the open, with course materials fully available to the public at no cost. The textbook is developed in open fashion. Open education brings multiple benefits. The curriculum author benefits from increased audience and impact. Instructors at other institutions benefit from an additional resource. Self-taught students benefit from the removal of gatekeepers.

Open resources are found at https://github.com/rbohrer/pl-course. Open auto-grading is an example of one moderate ungrading approach that could be used in HCPL-based courses: though the instructor sets a fixed evaluation standard, the student has full transparency into evaluation, allowing student-instructor interactions to focus on achieving course goals instead of grades.

In addition to the exercises described here, each chapter includes a list of proposed classroom activities. These activities include both active-learning work for students and presentational suggestions for instructors, to assist writing lecture plans.

4.3 Programming Languages Education

This project draws on a long line of PL education research, best summarized through prior textbooks. Though this project was undertaken because existing books were insufficient for the author's classroom needs, it still owes those books a great debt. We reuse proven approaches from prior texts, even as we radically challenge their scope and framing.

The works of Shriram Krishnamurthi are clear influences on the text. The author used Krishnamurthi's text *Programming Languages: Application and Interpretation* (PLAI) [24] prior to the development of her own text. We build on several insights of PLAI: implementation is appealing coursework for many computer science students, and frequent understanding checks throughout the text are essential. We de-emphasize programming paradigms [25]. The differences are substantial, however: we are language-agnostic, we include parsing, we limit implementation to a minimal language, and we include extensive interdisciplinary material. The integration of social issues within the book aligns with a growing trend of integration across the CS curriculum [8].

Not all our influences are textbooks. Courses offered by Coblenz and Aldrich [6] are major influences. We also cite non-text-books on programming languages, such as *Crafting Interpreters* [34], as a minor influence.

The HCPL text is developed with an awareness of successful texts on the theoretical foundations of programming languages [17, 35, 38], yet seeks to avoid duplicating them where they have already succeeded. On the contrary, the breadth approach aims to help identify when students wish to pursue theoretical foundations further.

Few textbooks address programming languages with academic rigor with methods from outside computer science. The most direct precursor on this front is *The Programmer's Brain* [18], but it is framed around programming in general rather than programming languages alone.

5 Discussion

This section explores several aspects of the book's practical use raised as discussion topics during peer-review.

How can educators with computer science backgrounds address teaching material outside CS?. Educators are encouraged to view their background as a strength rather than a weakness. In any interdisciplinary course, it is typical for the educator to have a stronger background in one discipline than others. Educators can draw on their own background to help tailor materials to suit their students. To ensure that educators are supported in this process, preparation can start from the provided materials, including the text itself as well as provided lecture slides. When students present with questions that an instructor cannot answer due to their background, the instructor can lean on the importance of self-regulated learning and encourage students to self-study by following relevant references in the book.

How can teaching goals outside CS be assessed? The authors' own courses reflect multiple strategies. The A-term 2021 iteration of CS 4536 included multiple-choice questions in weekly assignments. This format could be used to assess certain (e.g., factual) knowledge.

The Fall 2023 iterations of CS 536 and CS 4536 rely on ungraded peer assessment, with an intention of rewarding engaging while de-emphasizing the existence of a universal "correct" answer.

The HCPL textbook also includes essay-writing prompts, but these should only be used for evaluation when course staffs are equipped to evaluate writing.

What strategies can instructors take in using this book at scale? The book is designed with scale in mind, and the human-centered approach is not inherently harder to scale than others. In Fall 2023, it is used in two courses with over 80 combined students and a single (20-hour/week) teaching assistant. All programming work is auto-graded, which scales cleanly. Completion grading with peer evaluation is used for written work. This scales well, with the main barriers being (1) assignment of peer reviewers and (2) checking for bogus submissions. Tools exist which automate both of these tasks, achieving scalability. In particular, our course management system (Canvas) has built-in peer-assessment.

6 Conclusion

This paper presented the programming languages textbook *Human-Centered Programming Languages*, its design rationale, its pedagogical approach, and its potential applications in the classroom. This paper is written as the book reaches its initial public release. On one hand, the contents are not immature: the lecture notes which preceded the book have been used as the sole text for a semester-long course, and the book has expanded by a factor of two in that time, and undergone revision. On the other hand, revision will continue.

This paper includes a qualitative analysis of course evaluations from two PL courses and an introductory course using functional programming. The results of this analysis inform the ongoing process of revision. Through this process, the range of topics covered is likely to grow, but is never meant to replace PL depth courses, instead establishing a new form of PL breadth course founded in an interdisciplinary approach.

7 Acknowledgments

Thanks to Molly Feldman for encouraging me to submit this work to SPLASH-E. Thanks to Matthew Ahrens for feedback on a draft. Thanks to Gillian Smith, Chris Martens, Hannah Gommerstadt, and many others for encouraging me to keep writing the book, including the members of a semi-public Discord server[2] related to the book. Thanks to Amy Ko for developing the platform Bookish.press which was used to write the book. Human-Centered Programming Languages is the first textbook written on the platform other than Amy's own, and she has provided extensive technical support.

References

[1] Hugh H Benson. 2006. Plato's method of dialectic. In *Blackwell Companion to Plato*. Blackwell Publishing Malden, MA etc, Singapore, 85–99.

[2] Rose Bohrer. 2022. Imagining Introductory Rust. In *RustEdu*. Rust Edu, Virtual, 27–33.

[3] Rose Bohrer. 2023. *Human-Centered Programming Languages*. Self-published, Virtual. https://bookish.press/hcpl Author's note: it is possible that future editions will be published under an open-access academic press.

[4] Virginia Braun and Victoria Clarke. 2012. *Thematic analysis*. American Psychological Association, Washington, DC.

[5] Victoria Clarke, Virginia Braun, and Nikki Hayfield. 2015. Thematic analysis. *Qualitative psychology: A practical guide to research methods* 3 (2015), 222–248.

[6] Michael J. Coblenz, Ariel Davis, Megan Hofmann, Vivian Huang, Siyue Jin, Max Krieger, Kyle Liang, Brian Wei, Mengchen Sam Yong, and Jonathan Aldrich. 2020. User-Centered Programming Language Design: A Course-Based Case Study. *CoRR* abs/2011.07565 (2020), 7 pages. arXiv:2011.07565 https://arxiv.org/abs/2011.07565

[7] Michael J. Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2021. PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design. *ACM Trans. Comput. Hum. Interact.* 28, 4 (2021), 28:1–28:53. https://doi.org/10.1145/3452379

[8] Lena Cohen, Heila Precel, Harold Triedman, and Kathi Fisler. 2021. A New Model for Weaving Responsible Computing Into Courses Across the CS Curriculum. In *SIGCSE*. ACM, Virtual, 858–864.

[9] Haris Delić and Senad Bećirović. 2016. Socratic method as an approach to teaching. *European Researcher. Series A* 111, 10 (2016), 511–517.

[10] Carolyn Ellis, Tony E Adams, and Arthur P Bochner. 2011. Autoethnography: an overview. *Historical social research/Historische sozialforschung* 36, 4 (2011), 273–290.

[11] WPI CS Department Faculty. 2022. WPI CS Broadening Participation in Computing Plan.

[12] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to design programs: an introduction to programming and computing*. MIT Press, Cambridge, MA.

[13] Shaun Ferns, Robert Hickey, and Helen Williams. 2021. Ungrading, supporting our students through a pedagogy of care. *International Journal for Cross-Disciplinary Subjects in Education* 12, 2 (2021), 4500–4504.

[14] Interactive Fiction Technology Foundation. 2023. Twine 2.7.0.

[15] Susan E George. 2002. Learning and the reflective journal in computer science. In *Australasian Computer Science Conference*, Vol. 2. Australian Computer Society, Melbourne, 77–86.

[16] Rose M Giaconia and Larry V Hedges. 1982. Identifying features of effective open education. *Review of educational research* 52, 4 (1982), 579–602.

[17] Robert Harper. 2016. *Practical foundations for programming languages*. Cambridge University Press, Cambridge, UK.

[18] Felienne Hermans. 2021. *The Programmer's Brain: What every programmer needs to know about cognition*. Manning, Shelter Island, NY.

[19] Jan Hylén. 2006. Open educational resources: Opportunities and challenges. *Proceedings of open education* 4963 (2006), 10 pages.

[20] Steve Klabnik and Carol Nichols. 2023. *The Rust programming language*. No Starch Press, San Francisco.

[21] Amy J. Ko. 2016. What is a programming language, really?. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU@SPLASH 2016, Amsterdam, Netherlands, November 1, 2016*, Craig Anslow, Thomas D. LaToza, and Joshua Sunshine (Eds.). ACM, Amsterdam, 32–33. https://doi.org/10.1145/3001878.3001880

[22] Alfie Kohn and Susan D Blum. 2020. *Ungrading: Why rating students undermines learning (and what to do instead)*. West Virginia University Press, Morgantown.

[23] Dimitra Kokotsaki, Victoria Menzies, and Andy Wiggins. 2016. Project-based learning: A review of the literature. *Improving schools* 19, 3 (2016), 267–277.

[24] Shriram Krishnamurthi. 2007. *Programming languages: Application and interpretation*. Self-published, Online.

[25] Shriram Krishnamurthy. 2008. Teaching programming languages in a post-linnaean age. In *SIGPLAN Workshop on Programming Language Curriculum*. ACM, Cambridge, MA, 81–83.

[26] Imre Lakatos. 1963. *Proofs and refutations*. Nelson, London.

[27] T. Memmott. 2011. Codework: Phenomenology of an anti-genre. *Journal of Writing in Creative Practice* 4, 1 (2011), 93–105.

[28] John C Mitchell. 1996. *Foundations for programming languages*. Vol. 1. MIT Press, Cambridge, MA.

[29] Kristina MW Mitchell and Jonathan Martin. 2018. Gender bias in student evaluations. *PS: Political Science & Politics* 51, 3 (2018), 648–652.

[30] Brad A Myers, Amy J Ko, Thomas D LaToza, and YoungSeok Yoon. 2016. Programmers are users too: Human-centered methods for improving programming tools. *Computer* 49, 7 (2016), 44–52.

[31] Brad A. Myers, John F. Pane, and Amy J. Ko. 2004. Natural programming languages and environments. *Commun. ACM* 47, 9 (2004), 47–52. https://doi.org/10.1145/1015864.1015888

[2] Readers are welcome to contact the author for invitation to the server.

[32] Graham Nelson. 2001. *The Inform Designer's Manual*. Interactive Fiction Library, Virtual.
[33] Lene Nielsen. 2013. *Personas - User Focused Design*. Human-Computer Interaction Series, Vol. 15. Springer, London. https://doi.org/10.1007/978-1-4471-4084-9
[34] Robert Nystrom. 2021. *Crafting interpreters*. Genever Benning.
[35] Benjamin C Pierce. 2002. *Types and programming languages*. MIT Press, Cambridge, MA.
[36] Benjamin C Pierce. 2004. *Advanced topics in types and programming languages*. MIT Press, Cambridge, MA.
[37] Casey Reas and Ben Fry. 2006. Processing: programming for the media arts. *Ai & Society* 20 (2006), 526–538.
[38] John C Reynolds. 1998. *Theories of programming languages*. Cambridge University Press, Cambridge, UK.
[39] John Seely Brown and RP Adler. 2008. Open education, the long tail, and learning 2.0. *Educause review* 43, 1 (2008), 16–20.
[40] Stuart Shieber. 1985. Evidence againts the context-freeness of natural language. *Linguistics and Philosophy* 8 (1985), 333–343.
[41] Jesse Stommel. 2018. How to ungrade. https://www.jessestommel.com/how-to-ungrade/
[42] Alan Taylor. 1960. The flow-matic and math-matic automatic programming systems. *Annual Review in Automatic Programming* 1 (1960), 196–206.
[43] David Wiley, TJ Bliss, and Mary McEwen. 2014. Open educational resources: A review of the literature. In *Handbook of research on educational communications and technology*. Springer, New York, NY, 781–789.
[44] Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. 2020. Penrose: from mathematical notation to beautiful diagrams. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 144–1.
[45] Barry J Zimmerman. 2002. Becoming a self-regulated learner: An overview. *Theory into practice* 41, 2 (2002), 64–70.

Received July 27, 2023